TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MARKUS SINISALO

IMPROVING WEB USER INTERFACE TEST AUTOMATION IN CONTINUOUS INTEGRATION

Master of Science thesis

Examiner: Prof. Hannu-Matti Järvinen
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 13th January 2016

ABSTRACT

MARKUS SINISALO: Improving Web User Interface Test Automation in Continuous Integration
Tampere University of Technology
Master of Science Thesis, 83 pages
May 2016
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Hannu-Matti Järvinen

Keywords: software testing, test automation, continuous integration, web user interface

Modern software development makes use of agile methodologies in order to help responding rapidly to changing customer requirements and detected issues. Therefore, software can go through rapid changes in a short time. Changes to software may cause errors in previously functional features and these errors may be detected fast by automated regression testing. One method that addresses this issue is continuous integration that utilizes test automation as a part of the build. Thus, the developers receive rapid feedback and the basis of the software can be attempted to be kept stable.

A Finnish software company M-Files has developed web user interface test automation as a part of the quality assurance of its product. The goal of this thesis is to create improvement suggestions that can be used to solve or mitigate detected issues in the testing process and utilization of the aforementioned test automation tool. Additionally, the goal is to implement some of these suggestions. The main issues are the long duration of a test round and that many of the test automation related tasks are manual. Other detected issues are the lack of systematic monitoring of test duration, the lack of visibility to test results in the organization, and quality issues in the tests.

The literature review part of this thesis examined principles of continuous integration and how test automation can be utilized as a part of builds in continuous testing. Additionally, web user interface design, testing methods, and implementation technologies were discussed. In response to the identified issues in the web user interface test automation process and based on further analysis, 11 improvement suggestions were created. In the scope of this thesis, three of these suggestions were implemented and the implementation of another two suggestions was started. The implemented suggestions are as follows: automated web user interface tests were added as a part of a continuous integration build, test sets were divided into smaller components that support parallelism better, and a browser rotation principle for consecutive test runs was created. Additionally, a more precise quality control of tests was started, and an initial implementation for communication with M-Files REST API was created in the test automation tool so that test initialize operations could be streamlined in the future. As a result, more rapid feedback can now be gained from the web user interface test automation, the test results are more visible, and most of the tasks are automated as part of the build. Development of test automation continues in the organization, for instance, by further refining the results of this thesis.

TIIVISTELMÄ

MARKUS SINISALO: Web-käyttöliittymän automaatiotestauksen parantaminen jatkuvassa integroinnissa
Tampereen teknillinen yliopisto
Diplomityö, 83 sivua
Toukokuu 2016
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Hannu-Matti Järvinen

Avainsanat: ohjelmistotestaus, automaatiotestaus, jatkuva integrointi, web-käyttöliittymä

Nykyaikaisessa ohjelmistokehityksessä hyödynnetään ketteriä menetelmiä, joiden avulla yritetään nopeasti vastata asiakkaiden muuttuviin vaatimuksiin ja havaittuihin ongelmiin. Tästä syystä ohjelmisto voi käydä läpi nopeita muutoksia lyhyillä aikaväleillä. Muutokset ohjemistoon voivat aiheuttaa aikaisemmin toimiviin ominaisuuksiin virheitä, joita voidaan nopeasti havaita automatisoidun regressiotestauksen avulla. Eräs tähän ongelmaan vastaava menetelmä on jatkuva integrointi, johon kuuluu automaatiotestien liittäminen osaksi buildia. Siten kehittäjät saavat nopeaa palautetta ja ohjelmiston perusta voidaan jatkuvasti yrittää pitää stabiilina.

Suomalainen ohjelmistoyritys M-Files on kehittänyt web-käyttöliittymän automaatiotestausta osana tuotteensa laadunvarmistusta. Tämän työn tavoitteena on kehittää parannusehdotuksia, joiden avulla voidaan korjata tai lieventää kyseiseen testiautomaatioon liittyvän testausprosessin ja testaustyökalun hyödyntämisessä havaittuja ongelmia. Lisäksi tavoitteena on joidenkin parannusehdotusten toteuttaminen. Tärkeimmät havaitut ongelmat ovat testikierrokseen kuluvan ajan pituus ja useiden automaatioon liittyvien työvaiheiden manuaalisuus. Muita havaittuja ongelmia ovat testeihin kuluvan ajan systemaattisen mittauksen puuttuminen, testitulosten heikko näkyvyys organisaatiossa ja testeissa havaitut laatuongelmat.

Työn kirjallisuuskatsauksessa tarkasteltiin jatkuvan integraation periaatteita sekä testisautomaatiota osana buildeja jatkuvassa testauksessa. Lisäksi tutustuttiin web-käyttöliittymien kehitykseen, testausmenetelmiin, ja toteutusteknologioihin. Vastauksena web-käyttöliittymän automaatiotestauksessa havaittuihin ongelmiin ja tarkemman analyysin pohjalta luotiin 11 parannusehdotusta. Näistä ehdotuksista työn puitteissa toteutettiin kolme ja kahden toteuttaminen aloitettiin. Toteutukset olivat web-käyttöliittymätestien liittäminen osaksi jatkuvan integraation buildia, testisettien jakaminen rinnakkaisuutta paremmin tukeviin pienempiin kokonaisuuksiin ja peräkkäisissä testiajoissa käytettyjen selaimien kiertoperiaatteen luominen. Lisäksi testien laadun tarkempi seuranta aloitettiin ja testityökaluun luotiin alustava toteutus kommunikointiin M-Files REST-rajapinnan kanssa, jotta testien alustustoimenpiteitä voitaisiin jatkossa virtaviivaistaa. Työn tuloksena web-käyttöliittymän automaatiostestauksesta saatu palaute on nyt nopeampaa, testitulokset näkyvämpiä ja suurin osa työvaiheista on automatisoitu osaksi buildia. Testiautomaation kehittäminen organisaatiossa jatkuu muunmuassa jalostamalla tämän työn tuloksia eteenpäin.

PREFACE

First, I want to thank Minna Vallius and Tero Piirainen for their assistance in refining the subject for my thesis and allowing me to concentrate so much on the web user interface test automation in my work. Additional thanks to Minna for taking my wishes and interests so well in consideration when we were initially planning the thesis subject area, and for providing comments along the way. I also want to express my gratitude to all my colleagues in M-Files for their help with the work related to my thesis.

I also want to thank Professor Hannu-Matti Järvinen for his guidance regarding my thesis. Further, I want to express my gratitude to Matti Vuori for his insightful comments and valuable assistance. The various suggestions on how to improve my thesis have been most helpful.

I am also very grateful to my parents Hannele and Seppo for their continuous support during my studies in Tampere University of Technology and before. I also want to sincerely thank my sister Susanna and my brother Aleksi for their irreplaceable help during our journey together through the studies.

Tampere, 25.05.2016

Markus Sinisalo

TABLE OF CONTENTS

1. INTRODUCTION 1
2. TEST AUTOMATION IN CONTINUOUS INTEGRATION 3
2.1 Agile software development 3
2.2 Continuous integration 6
2.2.1 Common tools in continuous integration 6
2.2.2 Continuous integration practices 7
2.3 Basic approaches of software testing 12
2.4 Continuous testing and test automation 16
2.4.1 Test automation practices 16
2.4.2 Continuous integration builds in continuous testing 18
2.4.3 Build configurations and infrastructure considerations 20
3. WEB USER INTERFACE TESTING, DESIGN, AND IMPLEMENTATION 22
3.1 Design and testing of user interfaces 22
3.2 Web technologies 25
3.3 Practical methods in web user interface test automation 29
4. USER INTERFACE TEST AUTOMATION OF M-FILES WEB 33
4.1 M-Files product and M-Files Web 33
4.2 Product development process and testing in M-Files 36
4.3 Current state of user interface test automation 38
4.3.1 Technologies used in the user interface test automation tool 38
4.3.2 User interface test automation implementation 41
4.3.3 User interface test automation process 44
4.3.4 Identified points of improvement 46
5. IMPROVING WEB USER INTERFACE TEST AUTOMATION 52
5.1 Improvement suggestions and their prioritization 52
5.2 Implemented improvement suggestions 62
5.2.1 Adding automated tests as part of CI 62
5.2.2 Splitting test classes to support parallelism and clarity 68
5.2.3 Browser rotation for UI tests 71
5.2.4 Test quality control process improvements 71
5.2.5 Use of REST API in tests 72
5.3 Evaluation of the implemented improvement suggestions 73
5.4 Future thoughts and development ideas 75
6. CONCLUSIONS 77
REFERENCES 79

LIST OF ABBREVIATIONS

AJAX	Asynchronous JavaScript and XML
API	Application programming interface
AUT	Application under test
CI	Continuous integration
CSS	Cascading style sheets
DOM	Document object model
DSL	Domain-specific language
HTML	Hyper text markup language
IE	Internet Explorer web browser
IIS	Internet Information Services
JSON	JavaScript object notation
POM	Project object model
REST	Representational state transfer
UI	User interface
URL	Uniform resource locator
VC	Version control
XML	Extensible markup language

1. INTRODUCTION

Modern agile methodologies have caused acceleration in software development process. New features are developed in an iterative way, user requirements form a constantly moving target, and the code base is continuously modified. Nevertheless, product quality should not be compromised. This also sets a challenge to software testing: How to ensure product quality in a high-speed development process? It is evident that both manual testing and test automation practices have to be sufficient in order to catch defects and to prevent regression failures (Vuori 2014). Moreover, continuous and comprehensive testing activities are required throughout the development cycle because the product must remain in a stable state to build upon new functionality and features (Ariola 2015, Vuori 2014). Practices of continuous integration software development method, such as integrating automated tests to build process and fixing found errors as soon as possible, can also do their part in ensuring the product quality (Fowler 2006).

Development of test automation is often a software project in its own right (Kaner 2000). Building effective test automation is thus often also an iterative process that needs to be adjusted and improved based on observations and received feedback. The automated tests themselves must also be synchronized with the requirements of the product in order to gain accurate test results. Naturally, achieving this requires continuous maintenance work on the test automation.

Testing has also an important role in the product development process of a Finnish software company M-Files and the organization is continuously refining its quality assurance practices and tools. The tools range from test and error management to continuous integration and test automation. The organization has identified a set of problems in the current user interface test automation process considering the testing of M-Files Web, a browser-based application. First, the calendar time duration of running a full round of automated user interface tests is considered too long. This long testing duration hinders fast error detection and also makes it difficult to run the tests frequently. Moreover, many test environment setup tasks are done manually which consumes additional time and resources. A large amount of work is also invested in the analysis of the test results and manually repeating failed test cases. However, a significant number of these test failures cannot often be reproduced manually which indicates problems in the quality of the automated tests. Additionally, the current user interface test automation process does not emphasize visibility to test results and thus it is more difficult to judge the quality of a build or to make quick analysis on the results. Finally, the test durations are not systematically monitored which means that potential performance measurement data

is not utilized. This data could be used to detect changes in the performance of M-Files product or the test automation tool itself.

The goal of this thesis is first and foremost to make improvement suggestions that try to solve or mitigate the aforementioned issues in the web user interface test automation. Thus, the suggestions consider the test automation process, the implementation of the test automation tool, and the automated tests themselves. Finally, another goal is to implement some of the improvement suggestions that have the highest priority.

First, this thesis examines general approaches to software development and software testing before proceeding to the specific processes and practices in M-Files organization. Chapter 2 discusses agile software development and then continuous integration practices. The chapter also examines software testing and test automation, and how they relate to continuous integration builds. Chapter 3 presents practices in user interface design and testing, and also discusses web technologies that are used in implementing browser-based applications. Chapter 4 examines M-Files product including M-Files Web, M-Files organization, and its product development process. The chapter also presents the user interface test automation tool, the testing process related to it, and the identified points of improvement are discussed in detail. Chapter 5 presents improvement suggestions to solve or mitigate the issues in user interface test automation, and provides descriptions of the implemented suggestions. The chapter also discusses the evaluation of the implemented suggestions. Finally, chapter 6 presents conclusions of the thesis.

2. TEST AUTOMATION IN CONTINUOUS INTEGRATION

Development of software products has lately evolved into a rapid process. Software companies aim not only for fast deliveries to market but also for rapid responses to customer needs and identified problems (Vuori 2014). To achieve these goals, many of these companies have chosen to adopt agile software development practices and continuous integration principles as a part of their product development process. Utilizing automation in testing is one of the key practices in continuous integration (Fowler 2006). A set of automated tests that is run frequently can be a powerful tool in providing rapid feedback, and assurance of the quality, of the product in development.

2.1 Agile software development

Several agile software development methods have emerged from the need to better respond to changes and to rapidly deliver working software to customers. Many of these methods have their roots in the principles of the Agile Manifesto (Beck et al. 2001). The contents of the Agile Manifesto are as follows:

> *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> - *Individuals and interactions over processes and tools*
> - *Working software over comprehensive documentation*
> - *Customer collaboration over contract negotiation*
> - *Responding to change over following a plan*
>
> *That is, while there is value in the items on the right, we value the items on the left more.*

Therefore, as Beck et al. (2001) state, better results in software development are often achieved by good cooperation and communication between stakeholders, and being able to quickly respond to emerging changes in the project. Also, the focus of these agile methods is on working software. Thus, agile software methods tend to build the system iteratively, by gradually adding new features and functionality as the development progresses.

Thus, the iterative development in agile methods is based on small increments that are reviewed often by the customer. The duration of each development iteration is usually

2-8 weeks. This practice increases the likelihood that the customer requirements are fulfilled because they can regularly evaluate a working example of the product. Additionally, the customer can prioritize the functionality to be added to the product so that each regular increment brings more value to the customer. (Koch 2014 pp. 105-107)

Further, the iterative nature of development in agile methods provides the means to manage changes. The changes can be divided into external and internal changes (Koch 2004). External changes can often not be affected but should be reacted to. These changes can thus be, for instance, changes in standards that the customer must regulate against or new emerging technologies. Thus, external changes can pose new requirements but, on the other hand, can also offer new opportunities that can be seized. (Koch 2004, pp. 141-145)

Koch (2004) claims that internal changes emerge from the learning process that both the customer and the developers experience during the project. In the beginning of a project, the customer can rarely provide requirements that are complete, accurate, and realistic. However, the customer gains more knowledge and understanding on their requirements as the development of the system progresses. This learning process is possible only if the interaction with the customer is continuous in nature. Similarly, the developers may notice that the original plan has not taken some technical aspect into account, and thus a change in the implementation approach is required. (Koch 2004, pp. 141-145)

Therefore, agile methods do not encourage exhaustive planning before the project because both the customer and the developers will likely alter the requirements. On the other hand, the high-level project plan is revisited and refined before each iteration as more knowledge of the developed product is accumulated. Additionally, each iteration contains some detailed planning beforehand. (Koch 2004, p. 154)

Scrum (Schwaber & Sutherland 2013) is an example of an agile development method. The agile M-Files product development process has received influence from the Scrum method. Schwaber and Sutherland (2013) describe that development in Scrum method is based on the efforts of the Scrum team and regular Scrum events. This Scrum team consists of the product owner, the development team, and the Scrum master, each of which have their own role and responsibilities. The developed product's features, functionality, requirements, and needed changes in the requirements are expressed as items in an ever-evolving ordered list called the product backlog. The main event in Scrum is the sprint, an iteration with a maximum duration of a month, when the product is developed according to items selected to sprint backlog from the product backlog. Other events in Scrum are daily Scrum, sprint planning, sprint review, and sprint retrospective are events which respectively concentrate on daily communication, planning the work for the sprint, reviewing the results of the sprint, and inspecting the need to improve the working process of the team. (Schwaber & Sutherland 2013)

Naturally, agile software development includes multiple testing activities that have different goals. Some of these activities aim for fast detection of errors and some for assessing other parts of software quality, such as usability, security, and performance. Scrum, however, does not define any specific testing events but states that each increment must be thoroughly tested and must work with all previous increments (Schwaber & Sutherland 2013). A general approach to testing in agile development can however be described, for instance, by examining the testing process of M-Files organization combined with continuous integration practices. Thus, the Scrum method and how testing can be applied in it can be seen in Figure 2.1. Each increment contains items that are developed, such as new features, error corrections, or modifications of existing functionality. Each item may require changes to automated tests and developing new tests. These tests are executed to help ensuring that the implementations fulfill their specifications and that existing functionality is not unexpectedly affected by the changes. Additionally, manual testing is performed on new features in order to detect defects, identify points where the design is flawed, and otherwise assess the fulfillment of functional and non-functional requirements, as well as validate that the features suit their intended use.

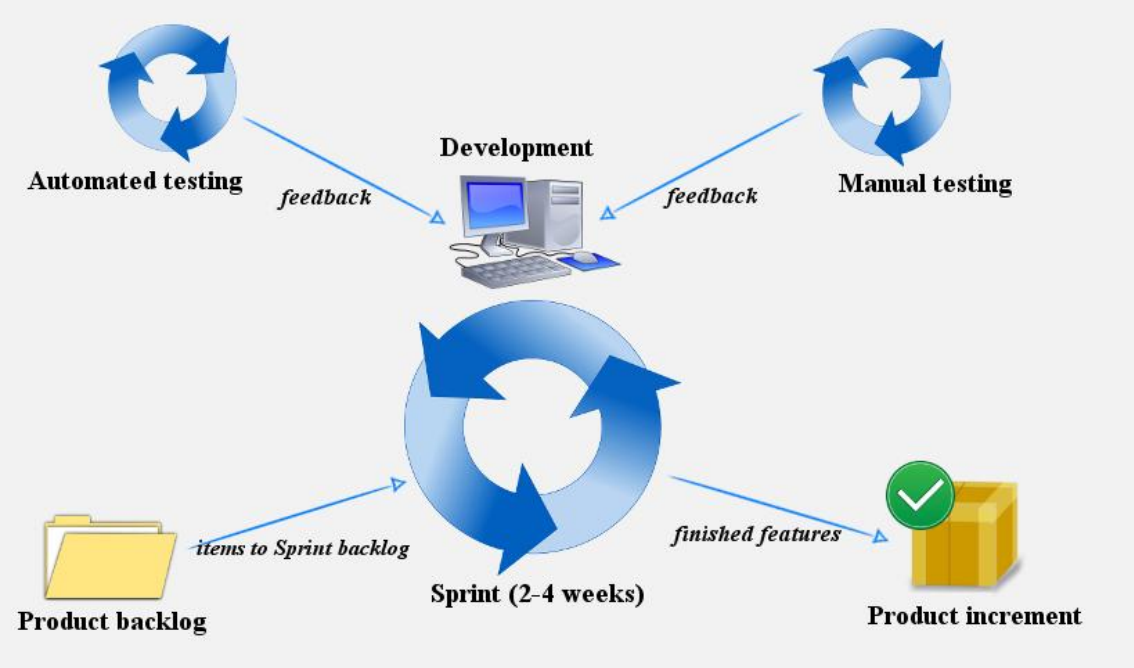

Figure 2.1 *Simplified process overview of agile software development and testing according to Scrum method*

In addition to Scrum, there are other software development methods that are considered agile, such as Extreme programming, Feature-driven development, and Lean software development (Koch 2004, pp. 7-8). However, the software development practices of an organization are often an adaptation of a single or multiple software development methods. Thus, M-Files organization has its own approach to Scrum method and continuous integration practices. That approach is discussed more in chapter 4.

It is thus evident that a software product developed by following principles of agile methods is likely to undergo multiple changes. Existing solutions may have to be modified in order to satisfy changed customer requirements and new features may require changes to previously implemented components. Additionally, such an ever evolving product requires the development team to constantly make decisions based on the current knowledge that they have available. With this nature of agile software development in consideration, this thesis next discusses continuous integration in more detail, and later, software testing practices including test automation.

2.2 Continuous integration

Continuous integration (CI) is a software development practice aiming for rapid detection and correction of issues, and thus providing essential support to agile and iterative development. It involves several key practices, such as maintaining source code in a single repository, automating builds and testing, the developers committing their changes frequently, and integrating the code continuously with others (Fowler 2006). Thus, working with CI usually requires the team to make use of different tools and technologies (Abdul & Fhang 2012) but the tools alone do not ensure a successful adoption of the process. Moreover, the team and all its individual members also have a personal responsibility to work according to the principles of CI in order to obtain the full benefits of the method (Abdul & Fhang 2012). Naturally, the responsibility concerns both the developers and the testers (Stolberg 2009).

The concept of build is important in CI. The build has a dual meaning that should be clarified before proceeding further into the practices of CI. First, a build is "an operational version of a system or component that incorporates a specified subset of the capabilities that the final product will provide" (ISO/IEC/IEEE 2010). Thus, a build is a development version of the software product that is usually identified by a build number (Techopedia.com 2016a). Second, build is the process which converts the source code into a form that can be run by a computer (Techopedia.com 2016a). This process can include testing and other, often automated, activities.

2.2.1 Common tools in continuous integration

Many organizations choose to implement CI with the help of two specific tools: version control (VC) and continuous integration server (CI server). Next, these tools are briefly discussed, and then the process itself and its key practices are presented.

Version control

Version control, is a system that keeps record of modifications made to files. Two common types of VCs are centralized version control systems and distributed version control systems. In both cases, the version controlled files are stored in a repository in a

server computer. Thus, different users can access the VC server by using clients to check out these files. The user can then modify the files and make these updates available to other users by committing the changes to the server. Additionally, distributed version control systems have the whole repository mirrored to the client computers. Thus, distributed version control provides an extra layer of safety to recover from server failure. (Chacon & Straub 2014, pp. 27-30)

VC generally allows the user to revert single or multiple files to their previous version, see changes between versions, and see who has made those changes, for example. (Chacon & Straub 2014, p. 27) Thus, VC is a highly useful tool for collaboration in software development projects. For instance, two common version control systems are Subversion (Apache.org 2016a), which is also known as SVN, and Git (Git-scm.org 2016).

Continuous integration server

CI server is an application that many teams find essential in the CI process (Fowler 2006). Typically the server monitors the VC and starts a build when it detects a commit. CI server can run scripts to perform different tasks, such as build the system under development and run automated tests (Enos 2013). For example, Teamcity (Jetbrains.com 2016a) and Jenkins (Jenkins-ci.org 2015) are both commonly used CI servers.

Teamcity, as an example of a CI server, offers features, such as automatic and manual triggering of builds, instant notifications to interested parties about build results, code changes comparison, and configurable test reports. Further, the user can configure version control settings in a build configuration and then define additional build steps. User selects a build runner for each step, such as Maven or PowerShell, to enable running scripts of that runner type. Thus, the build is run step by step by executing the user-defined steps. The build may be configured to produce files called build artifacts, such as coverage data or compiled binaries. These build artifacts may be presented to be downloaded or they can be used in other builds. (Melymuka 2012, pp. 8-17)

2.2.2 Continuous integration practices

According to Fowler (2006), developing software with CI involves running automated builds in different stages of the daily development work, regular integration with others' work, and reacting to issues as quickly as possible. Thus, working with CI can be described by a process containing the following steps (Fowler 2006):

1. Checking out the latest sources from VC as a working copy
2. Making modifications to the sources
3. Making an automated build on the local development machine (and fixing the build if there are any errors)
4. Updating the working copy with changes made by the other developers

5. Making an automated build on the local development machine (and fixing the build if there are any errors)
6. Committing changes to the VC
7. Making an automated build on the integration machine
8. Results of the build are distributed and communicated to everyone in the team (and the build has to be fixed if there are any errors)

The same steps are visible in Figure 2.2. The figure depicts a CI process in practice from a developer's point of view.

Figure 2.2 *Developing software in practice according to continuous integration process as described by Fowler (2006).*

Steps from 2 to 5 are iterated as long as it takes to make a successful build locally. Step 7a in Figure 2.2 signifies that the integration server makes a checkout from the VC repository before proceeding to step 7b. It should also be noted that the process basically starts again from step 1 if the integration build fails in step 7.

Key practices of continuous integration

This presented way of working is based on the key practices of continuous integration. These practices provide a more detailed overview on the process. According to Fowler (2006), effective CI is based on the following key practices:

1. Maintaining a single source repository
2. Automating the build
3. Making the build self-testing
4. Everyone committing to the mainline every day
5. Every commit building the mainline on an integration machine

6. Fixing broken builds immediately
7. Keeping the build fast
8. Testing in a clone of the production environment
9. Making it easy for anyone to get the latest executable
10. Everyone being able to see what is happening
11. Automating deployment

These practices form the basis of CI in the form how Fowler presents it. Next, the practices are presented in more detail based on Fowler's (2006) description.

1. Maintaining a single source repository: The first practice requires that the team uses a VC tool. The repository essentially contains everything that is needed to make the build. Thus, everything from source code, test scripts and configuration files to third party libraries are stored in the VC to avoid any problems in obtaining the dependencies for the project (Fowler 2006, Abdul & Fhang 2012). Additionally, VC can be used to control other files that the team works with but which may not be required by the build. Further, most VC systems enable the creation of development branches. Most of the development takes place in the mainline but the team can also utilize different branches on temporary experimenting on new features and providing bug fixes for the previous releases.

2. Automating the build: Creating the build is a crucial part in the CI process (Abdul & Fhang 2012). Therefore, the second practice, automating the build, essentially means that the build process can be abstracted into a single command that automatically results in a running system on the machine. Thus, automating the build requires utilization of build scripts that can build the system from the source files checked out from the VC. However, the build system should allow its user to control the build to produce alternative results for different situations. These alternatives include, for instance, building the system with or without tests, building the system with different combinations of test sets, or building an independent stand-alone component. In practice, automating the build helps the team to reduce mistakes in the build process and to make it easier for anyone to bring the system under development up and running.

3. Making the build self-testing: This practice involves the development of automated tests that are then run as a part of the build process. These automated tests should be utilized to test a large part of the code base to find defects. Similar to the whole build process, anyone should be able to launch the tests by issuing a simple command. Essentially, any failing test should also cause the build to fail. Different aspects of software testing, test automation, and utilizing test automation in the CI process is discussed in more detail later in this thesis.

4. Everyone committing to the mainline every day: This practice implies that the developers frequently integrate their work with the others. Thus, the developers should make their commit at least once a day but more frequent commits can be even more benefi-

cial. As a good practice, the developers should be able to build the system passing all the tests on their local machine before making the commit to the repository mainline. Frequent commits make it faster to detect any issues in the integration and thus also enable faster fixing of the issues. Additionally, integrating the code and resolving any conflicts is also easier because there are likely to be less commits made by other developers in this shorter time frame (Fowler 2006, Abdul & Fhang 2012).

5. Every commit building the mainline on an integration machine: It is important that a developed application is not dependent on any unique configuration or setup of the development computer (Abdul & Fhang 2012). Thus, the build should also succeed on an integration machine in addition to the successful build on the developer's own local computer. This practice effectively means that the build is triggered on a server machine after each commit made by a developer. The build can be started automatically by using VC monitoring with a CI server or it can be started manually. Nevertheless, this build serves as a confirmation that the build really succeeds in an environment that is different from the development environment. Additionally, this practice makes sure that the developer who made the commit has followed the previous practice, that is, passing the build successfully on the local development machine. In practice, the developer should monitor the progress of the build on the integration machine and be ready to address any issues that may arise.

6. Fixing broken builds immediately: Successful implementation of CI also requires that all team members take responsibility for keeping the build intact, that is, the source code compiles without errors and all the tests are passed (Abdul & Fhang 2012, Gmeiner et al. 2015). As a consequence, this provides a stable basis for the development. Thus, fixing broken builds immediately is a practice that should be on a high priority. Fixing the build should preferably take place on a development machine and the latest commit should be reverted so that the mainline remains stable.

7. Keeping the build fast: CI aims to provide quick feedback to the developers. Therefore, it is important that the team follows the practice of keeping the build fast. The specification of a fast build may vary depending on the project and the system under development but, in general, a build taking over an hour to finish can be considered too slow to provide the agility needed for a proper CI process. However, usually it is the tests that make up most of the duration of the build. Thus, the team can split the build into a sequence of builds that each perform different tests on the system under development. The first build of the sequence, that may be called the ”commit build”, should be able to finish quickly but still provide a reasonably adequate statement if the build can be considered as stable. This kind of build sequencing and different alternatives for setting up testing for them is discussed in more detail later in this thesis.

8. Testing in a clone of the production environment: The used environment may have a great impact on the system behavior and some errors may only appear in certain envi-

ronments (Gmeiner et al. 2015). Thus, the development team must also consider how the system will behave in the production environment of the end users. Further, testing should take place in a clone of the production environment considering variables such as the operating system, database software, third party software and network setup (Fowler 2006, Vuori 2014). All differences between the testing environment and the production environment add uncertainty of how the system will behave in the end use. Naturally, all different environments can often not be tested because the number of different combinations of variables grows too large. The role of the testing environments in CI and test automation is discussed in more detail later in this thesis.

9. Making it easy for anyone to get the latest executable: This practice is one way to help receiving rapid feedback. Simply, there should be a commonly agreed location where anyone can find the latest executable. This executable can be tried out by anyone and they can provide feedback on how it works, looks, and feels, and if something should perhaps be changed. Naturally, this common location is also useful for test automation, manual testing, and other activities that require the latest build.

10. Everyone being able to see what is happening: It is important that the current state of the system under development is communicated to everyone. Thus, this practice requires an implementation of some method to monitor the status and changes of the recent builds. Further, this information must be made visible for everyone. The aforementioned CI servers often provide a web page containing such build information. A web based approach is especially useful if the team members are working in different locations. Also, an automated issue tracking mechanism can be used to deliver the build status to the team (Abdul & Fhang 2012) or the team members can be notified by automatic emails. In general, the indicators for build stability can be anything that simply makes the information easily visible.

11. Automating deployment: As mentioned before, deploying the executable on different testing environments is common in the CI process. Thus, automating deployment brings many benefits to the team. In general, automatic deployment is faster and less prone to errors when compared to doing the work manually. Automatic deployment can usually be achieved by a script. Additionally, scripts can be tailored for deployment into the production environment but also for rolling back the deployment. Discussion on the automated deployment and its relation to the test automation is continued in later in this thesis.

Continuous integration shaped by the organization

The main benefit from continuous integration is that it continuously provides the most recent, fully integrated system that can be tested by both build-integrated test automation and by other means of testing. Additionally, the frequent, small commits make it

easier to locate the possible causes for detected issues. Of course, finding the defects is dependent on the testing process of the organization and the quality of the tests.

The practices presented in this chapter provide a high level description of working with continuous integration in software development. Each organization implements their own CI practices depending on their development projects, products in development, organizational structure, customers, and other internal and external factors. CI by itself does not solve everything. The process concentrates on the day to day development work and revolves around the concept of build and automated testing. Thus, it is also important to have a clear vision on the more high level direction of the development efforts towards the next release, and fulfilling the business requirements and user needs of the product in development. Software testing, its different practices and techniques, can help in achieving these goals which is discussed more next. On the other hand, M-Files organization's own approach to CI and other aspects of product development process is described later in chapter 4.

2.3 Basic approaches of software testing

Software testing has different aspects and thus there are many ways to define it. For instance, software testing can be defined as a process of executing a program with the intent of finding errors (Myers et al. 2011, p. 6). Another definition of testing is that it is an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component (IEEE Computer Society 2008). Further, testing can provide quality information of the product and aid in central decision making in the development cycle (Ariola 2015).

On the other hand, testing can also provide information on how different variants are preferred by users. This type of A/B testing presents the users with different versions of the product and data is gathered on which version is most preferred or effective (Techopedia.com 2016b). Additionally, testing of usability considers information gained by observing users in realistic work environment when they perform tasks by using the application (Galitz 2002, pp. 718-719). This type of testing does not only detect errors but also confusing design or causes for frustration.

Testing can also help in preventing the accumulation of technical debt in the developed system (Ariola 2015). This debt means suboptimal design choices that may result in faster implementation of functionality but which create problems later in development (Cunningham 1992, Fowler 2003). In practice, technical debt often also introduces defects into the system (Humble & Farley 2010, p. 330). An accelerated development process may generate technical debt fast if the development practices and the state of the product are not sufficiently monitored, for instance, by testing activities (Vuori 2014).

Software testing has also two aspects that are used in the testing terminology: verification and validation. Verification attempts to evaluate that the product fulfills the conditions set to it in a particular development phase (ISO/IEC/IEEE 2010). On the other hand, validation confirms that the application fulfills the requirements of its intended use and satisfies user needs (ISO/IEC/IEEE 2010). In other words, verification attempts to ensure that the product is developed into the direction as it is specified. Validation, on the other hand, attempts to make sure that the end users will be able to achieve their goals by using the application.

As it can be noted, testing has many aspects to it, but so too has product quality. One example of different quality levels is presented by Vuori (2014):

- Safe and secure
- Deployable
- Useful
- Technically solid
- Usable
- Good user experience
- Desirable

The quality levels can have different priorities depending on the developed product, its target audience, and intended use (Vuori 2014). Moreover, different testing techniques have to be applied in order to properly assess each of these levels in the product.

Testing approaches and levels

Software testing can be approached from two different angles based on how the AUT is perceived: black-box testing and white-box testing. Black-box testing technique is based on inputs and the corresponding outputs of the system, and whether they are aligned with the system's specification. Thus, black-box testing does not use or require any knowledge of the internal structure of the system. On the other hand, white-box testing makes use of examination of the internal logic of the program, that is, the code. With this approach, the tests may be designed to explore the control flow of the program more thoroughly. However, the number of different input combinations and unique execution paths in any non-trivial application is often infinite. Thus, exhaustive testing using either of these approaches is practically often impossible. (Myers et al. 2011, pp. 8-12)

Software testing is often structured into different levels. It should be noted that slight variations exist in the naming and definitions of these testing levels. Hass (2008), for instance, defines these levels as acceptance (or user acceptance) testing, system testing, integration testing, and unit (or component) testing. These testing levels are also commonly used in the context of the so called software development V-model. According to the model, each of these testing levels corresponds to a specific level of design in the

software development process. Thus, acceptance testing corresponds to the user requirements, system testing corresponds to the system requirements and specification, integration testing corresponds to the architectural design, and unit testing corresponds to the detailed design of the modules. In essence, the V-model is a sequential model and thus different adaptations have emerged for the needs of more iterative and incremental development processes. For instance, some testing activities from all testing levels can be carried out in a single development iteration. (Hass 2008, pp. 3-8)

The goal of unit testing is to find defects in the implementation of individual software components. Individual components are classes, functions, or other distinguishable units in the software, depending on how the organization wants to define a component. Nevertheless, unit testing considers the component in isolation of the other components. The isolation can be achieved by using test drivers to execute the component and stubs to simulate other components. Both white-box and black-box techniques can be used in unit testing to ensure sufficient structural coverage as well as the fulfillment of the specification. Unit testing of a component is mostly the responsibility of its developer. (Hass 2008, pp. 9-11)

Integration testing, on the other hand, considers assessing functionality that requires interaction between different components in the application. These tests, for instance, may be able to detect errors in the coordination of object and data lifecycles in the AUT. As opposed to unit testing, integration tests often utilize real databases, file systems, and other dependent systems, and may thus also detect errors in these areas of compatibility. (Humble & Farley 2010, p. 89)

System testing is done on a system with all its components fully integrated. The testing covers both the functional and non-functional requirements set to the system (Hass 2008, p. 14). Thus, areas such as usability, security, performance, compatibility, installation, documentation, and more, should be considered in system testing (Myers et al. 2011, p. 122). Also, experience-based techniques, such as exploratory testing, can be incorporated to system testing. Design of good system test cases often requires creativity and experience from the tester. (Hass 2008, pp. 14-15; Myers et al. 2011, p. 122)

Acceptance testing is done in close cooperation with the customer of the product. This testing assures that the complete product delivers everything according to agreements. Reported results of successful acceptance testing are required as evidence that the product fulfills the agreed acceptance criteria. In some cases, the customer may have full responsibility for the acceptance testing of the product. (Hass 2008, pp. 15-16)

Additionally, the aforementioned exploratory testing is an experience based testing technique that augments other more systematic testing practices. Exploratory testing is practically simultaneous test design and test execution which means that new testing paths are executed based on the observations made by the tester. Effective exploratory

testing requires good knowledge on different testing techniques and typical errors, and also domain knowledge of the application may be beneficial. These testing sessions should be documented, notes should be taken, and the findings are reported. The scope of exploratory testing session can also be narrowed down, for instance, by assuming a certain user role or concentrating on certain tasks. (Hass 2008, pp. 218-221)

One basic form of software testing is regression testing. This testing is performed on a system to verify that modifications and maintenance have not caused defects on previously working functionality (ISO/IEC/IEEE 2010). Moreover, regression testing can act as a safety net to discover bugs introduced to the system when the code is changed (Ariola 2015, Vuori 2014). Effectively, regression testing is a category that considers all aforementioned levels of testing. Thus, practically all automated tests can be run again as regression tests (Humble & Farley 2010, p. 87). Regression testing can also be focus on certain functionality or modules depending on the nature of the changes (Vuori 2014). In practice, regression testing is a repetitive task that considers executing existing tests again after the application code has been modified. This type of repetitive testing is suitable to be automated which leaves human testers with more time to focus on tasks with higher value, such as exploratory testing, manual validation, and testing of usability (Humble & Farley, p. 128).

Static quality assurance practices

There are also quality assurance methods that do not necessarily involve execution of the code. For instance, code inspections and walkthroughs aim for error detection by reading the code. These sessions usually have about four participants including the developer of the code that is being inspected. Often the aim is to detect errors and error-prone solutions while leaving the solving of them for later. The participants should prepare themselves by getting acquainted with the code and other material, such as design specifications. Also, a checklist of common errors is often used to guide the process. (Myers et al. 2011, pp. 22-37)

Automatic static analysis can prevent a large number of software defects from being introduced in the first place. Such analysis can include, for instance, checking code style and detecting syntax errors in the code. Further, this prevention saves allocating resources to error diagnosing, defect fixing, and re-testing the fixed update. (Ariola 2015, Vuori 2014)

On the other hand, organization's policies can also promote actions that improve product quality. Preferably these actions should be continuous practices that can be monitored as opposed to actions that only react to detected incidents. The policies may consider the aspects of how defects are introduced to the product, identify high risk activities that require defect prevention, and how to detect the defects as early as possible. (Ariola 2015)

2.4 Continuous testing and test automation

Agility in software development tends to cause different types of acceleration in the process. Thus, new products might be released in a fast continuous pace. On the other hand, new features might be continuously published and thus producing steadily more value to the customer. (Vuori 2014)

The acceleration in the process often makes testing as the bottleneck of the development cycle. This is especially the case if testing is regarded as a fixed event before the release of the product (Ariola 2015). However, it is testing that can provide more control in a rapid development process and bring more visibility about the current situation (Vuori 2014).

Similar to continuous integration, also continuous testing considers the ways of working in an organization rather than only the utilization of different technologies and tools. Successful continuous testing requires test automation but also the assessment of working methods and processes related to software development and quality (Ariola 2015).

2.4.1 Test automation practices

Test automation comes with its benefits and costs and these should be taken into account when considering the automation of a test. First of all, it should be noted that test automation will not catch all defects and should be augmented with manual testing (Vuori 2014). For instance, automated tests cannot practically verify good usability, and consistent look and feel, or perform effective exploratory testing (Humble & Farley 2010, p. 87). On the other hand, test automation is suitable for repeatable regression testing and thus neglecting this aspect can cause regression defects go undetected (Marick 1998, Vuori 2014). However, it should also be noted that automated testing often requires manual work in many of its phases (Kaner 2000). For example, designing the tests, debugging the tests, and reporting the errors are usually done manually. Further, the test results often require manual failure analysis.

According to Marick (1998), the team should compare the cost of automating a test case and running it once with the cost of running it once manually. For example, some defects might not be detected if resources are spent on test automation instead of manual testing. Moreover, the expected lifetime of the test should be estimated and it should be analyzed whether the investment to it will be returned during that lifetime. Finally, the team should consider the likelihood that the test will find additional defects in automation during its lifetime. (Marick 1998)

Design and development of automated tests

Optimally, all individual tests should be traceable to business requirements of the product. For instance, peer reviews done by other testers and developers may prove useful in validating the test cases. A well designed automated test suite can be continuously executed in repetition, and it will raise awareness of the impacts of code changes to these requirements and helps detecting regression defects as early as possible. Similarly, it is beneficial that the tests are logically correlated to the product's components and thus change impacts and test failures are easier to analyze. Additionally, tests should be deterministic and thus a test should only fail when an actual problem is detected. The reason for this failure must then be clearly communicated by the test. Similarly, a passing test should have an unambiguous meaning. (Ariola 2015)

Automated tests should be developed both on the lower unit level and the higher system level. The unit tests are typically faster to run and thus enable faster feedback. Additionally, unit tests encourage better design decisions, help in ensuring a stable code base, and make refactoring less daunting (Gmeiner et al. 2015, Vuori 2014). Another point of view to the subject is test driven development (TDD), a method to develop the automated tests before coding the feature and then gradually make all the tests pass as the feature unfolds (Fowler 2006). Writing the unit tests beforehand often gives the developer a better understanding of the developed feature and its specification (Myers et al. 2011, pp. 185-186).

On the other hand, system level regression tests should also be automated because running them manually consumes plenty of resources (Stolberg 2009). Additionally, some of these automated tests can simulate common user tasks or scenarios (Marick 1998). These tests contain interaction between multiple features and are likely to detect the same errors that the users would encounter. However, these tests are more vulnerable to changes in the application because of the wide variety of features they depend on. Development of features and automated tests for them can be carried out in parallel by multiple developers and testers (Stolberg 2009).

Maintenance of automated tests

As mentioned before, the VC should also contain all code, scripts and configurations needed by test automation. Of course, the first reason for the test sources being in the VC is that the tests can be executed as part of the build. But another as important part is that the tests must be maintained and new tests are required to be added to the test suite along with any changes to the system under development (Fowler 2006, Stolberg 2009).

Test maintenance is optimally performed as soon as a new business requirement is implemented or an existing one is changed. Otherwise, outdated tests are likely to cause defects get past testing undetected or cause unnecessary failures, also known as false positives, that have to be analyzed (Ariola 2015). However, successful maintenance

requires good communication between the developers and the test automation development team about the modifications (Gmeiner et al. 2015). In practice, maintenance of existing tests simply considers if some of the following actions should be taken: deleting the test, updating the test, updating the test assertions, updating the test data, and updating the test metadata (Ariola 2015). It is especially important that the value of a test is assessed again when it requires maintenance (Marick 1998). These maintenance tasks should also be prioritized (Ariola 2015).

When a test does fail the team should follow a certain workflow to analyze the failure. The workflow typically involves tasks that are assigned to according team members (Ariola 2015). This failure analysis requires effort but it is necessary in order to keep the tests synchronized with the AUT. It is especially vital that failing test cases are not commented out of the test code but that real work is put into finding out the causes for the failures, whether they are caused by regression defects or outdated assumptions about the functionality (Humble & Farley 2010, p. 70).

Test maintenance work can be laborious so it is necessary to design a flexible architecture for the test automation system (Gmeiner et al. 2015). Marick (1998) presents an idea how tests can be made more maintainable by utilizing application specific test libraries. Thus, Marick suggests that an automated test interacts with three layers of code: test libraries, intervening code, and code under test. The code under test is the code that implements the functionality that is being tested. On the other hand, the intervening code is application code calling the code under test. Thus, the intervening code is usually an user interface or an API (application programming interface). Finally, test tool and libraries layer is built on top of the intervening code for the purposes of the actual test code. The test libraries filter out the detailed implementation of the intervening code and allow the test to exchange condensed input and output with it. Therefore, changes to the intervening code should only require the team to update the test libraries and not to the actual tests. (Marick 1998) Additionally, the testability of the AUT can be improved by enhancing the APIs with additional interfaces, providing access to data or operations that are useful in the testing (Stolberg 2009, Gmeiner et al. 2015).

2.4.2 Continuous integration builds in continuous testing

Automated builds are in the core of continuous integration. These builds can be utilized both in compiling the application and executing related automated tests. For instance, Humble & Farley (2010) present a software development process that utilizes different CI builds in order to achieve a continuous flow of rapid feedback. Moreover, their methodology, called deployment pipeline, emphasizes the importance of developing software with a clear process that is based on the strengths of both manual and automated testing. In this pipeline, a build goes through a series of stages, each of which contains different testing activities. Each stage provides feedback to the developers and, should the stage succeed, increases confidence in the readiness of the build (Humble &

Farley 2010, pp.106-113). A very simple version of a build pipeline with two stages can be seen in Figure 2.3.

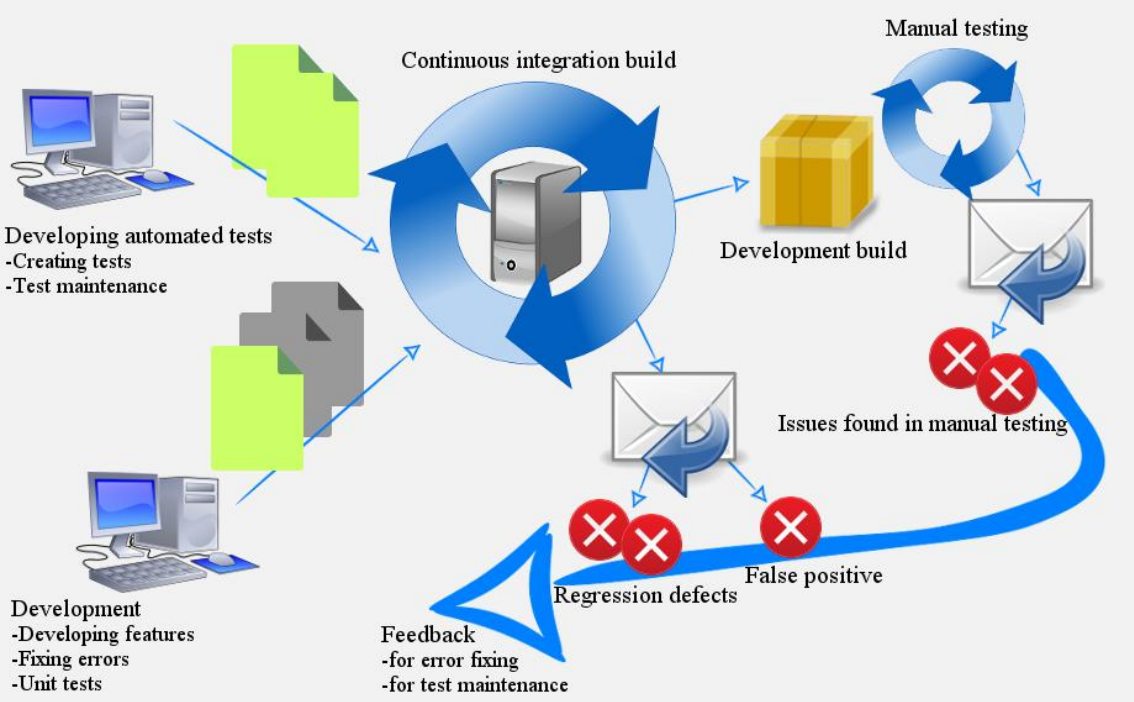

Figure 2.3 *A simple example of a build pipeline, with two stages, that supports iterative development: Continuous integration build stage and Manual testing stage. Both stages provide feedback to developers. In addition to detecting regression defects the CI build stage may also cause false positives that require test maintenance actions.*

It is essential that different forms of testing support each other and that no areas are neglected. Humble and Farley (2010) support this idea with their pipeline stages, by gradually putting the build through different types of testing and making the testing environment more production-like. Their first stage, the commit stage, compiles the software, executes unit tests, performs static code analysis, and stores compiled binaries to a common repository for an easy access. In order to keep the initial commit build stable, defects that are caught there have often the highest priority to be fixed (Fowler 2006). The commit build should also be run by the developers in their own environment before they make their commit. (Humble & Farley 2010, pp. 106-113)

A passing commit stage in the integration machine will progress to next stage that configures a testing environment, deploys the application, and executes automated system level tests. After successfully passing this stage, the build will be manually tested. This testing may thus include exploratory testing, and testing of usability, security, and performance. Both configuring the testing environments and deploying the build may be done with the help of automated scripts. Each stage provides feedback to the developers and, typically, the feedback from the first stages is provided faster. The first obvious reason for this is that the stages are sequential but the other reason is that the automated

tests are typically the faster to execute the lower level they are developed. Finally, a build that passes all stages may be deployed to production-like staging environment or into actual production use, again, with the help of automation. (Humble & Farley 2010, pp. 106-113)

It should be noted that the pipeline by Humble & Farley (2010) is one example of how software can be developed by utilizing manual testing supported by continuous integration and other automated activities. The CI build structure will be shaped by the developed software and the organization's processes. For instance, Gmeiner et al. (2015) report good improvements on their software development process by implementing a similar pipeline but with modifications that suit their project. Their pipeline has two parallel branches that support their software architecture: one for server backend and its desktop clients, and the other for a web application with its own backend. Additionally, their commit stage also contains automated integration tests in order to fill in gaps in unit testing. Naturally, also the maintenance of automated test suites is very important in order to receive reliable results by the test builds. Additionally, augmenting the earlier builds with some further tests should be considered if a defect gets past one stage but is then detected in later stages. (Fowler 2006)

Other variations of the process could include more stages or the stages may not be so tightly gated. For instance, a build may contain commits by two developers and the other commit caused some regression defects. On the other hand, the other parts of the application may be intact. Therefore, it may make sense to perform manual exploratory testing on the feature implemented by the other commit, as long as any of the regression defects in the build do not prevent the use of that functionality. Naturally, if a new build without the regression defect is quickly available then that should be used.

2.4.3 Build configurations and infrastructure considerations

Automated CI builds with integrated testing requires multiple tools, such as VC, CI server, test runners and environments. Thus, it has to be considered how the required set of tools will be hosted on the infrastructure. Multiple tools hosted on a single computer can cause performance issues if the processes are running concurrently. Therefore, both the memory consumption and CPU usage have to be taken into account (Stolberg 2009). Additionally, available disk space must be considered and monitored to avoid failures in the process. However, distributing the tools to multiple computers can cause network delay and may increase the vulnerability to network issues. (Abdul & Fhang 2012)

As mentioned before, one of the characteristics of CI is the automated testing in an environment similar to the production environment (Fowler 2006). Usually the production environment is composed of multiple different variables, such as operating system and third party software and therefore the production environment can differ between the end user organizations and also between individual users. Therefore, the team often has

to set up multiple different testing environments to ensure the system's compatibility with different environments.

One method to set up the required machines and environments is by making use of virtual machines (Fowler 2006, Vuori 2014). Thus, a single physical computer can host multiple virtualized machines each having its own configuration. However, the aforementioned considerations for tool distribution and disk requirements still apply for the virtual machines because they are utilizing the resources of a physical host machine.

Also test data management should be considered. Utilization of realistic test data, and easy access to it, can help the tests give more accurate results. This data should be reusable across teams, product versions, and releases. An example of good test data may be a copy of a production database which has its privacy and security related information cleansed. (Ariola 2015)

It is also beneficial to define the build configuration to be flexible and suited for long-term use. Dividing the build configuration to modular steps helps in modifying the build by adding, removing or modifying the steps, or re-using a step in another build configuration. Additionally, sufficiently flexible build configuration can be utilized in different environments with only minimal changes. (Abdul & Fhang 2012)

3. WEB USER INTERFACE TESTING, DESIGN, AND IMPLEMENTATION

This chapter discusses user interface design and testing in order to provide a broader understanding of the area where UI test automation operates. Moreover, the chapter examines web technologies that are involved in the implementation of web user interfaces of browser-based applications. Finally, some practical methods for implementing web user interface test automation are presented.

3.1 Design and testing of user interfaces

User interface is the only part of the application that most users interact with. It provides the users with access to the system's functionality but also has an important role in how they experience the product and feel about it. Therefore, user interface design is an important part of software development.

The concepts of user experience and usability are often used in the context of UI design. Norman and Nielsen (2016) present a definition for user experience that has many aspects:

> *The first requirement for an exemplary user experience is to meet the exact needs of the customer, without fuss or bother. Next comes simplicity and elegance that produce products that are a joy to own, a joy to use. True user experience goes far beyond giving customers what they say they want, or providing checklist features. In order to achieve high-quality user experience in a company's offerings there must be a seamless merging of the services of multiple disciplines, including engineering, marketing, graphical and industrial design, and interface design.*

On the other hand, Nielsen (2012) defines usability as a quality characteristic of a user interface that is composed of five different quality components: how easy it is to learn to use, how efficient it is to use, how easily the use can be memorized and later returned to, how many errors users make, how severe these errors are, and how easy it is to recover from these errors, and finally, how pleasant it is to use the design. Thus, as Nielsen and Norman (2016) state, user experience is a broader concept than UI design and design for good usability. It is evident that both user interface design and usability are important in user experience design. In addition to that, their definition of user experience also considers the whole experience of interacting with the company, its services,

and its products, and that the user really benefits from the offered solution (Norman & Nielsen 2016). A slightly narrower view on user experience may also be taken by mainly considering only the aspects of interaction between the user and the product (Brown 2013, p. 40). Naturally, the offerings and services of the whole company should provide the user with good experience but many of those matters often fall out of scope of what can be achieved with UI design of the product.

User interface and user experience design in agile development

Galitz (2002) presents user interface design process as a series of activities that require expertise in different areas, such as human behavior, visual design, usability and software development. These activities include, among others, choosing suitable user interface components, providing feedback to the user, choosing used colors, and organizing the UI layout. However, Galitz states that the first steps in this design are knowing and understanding the user, and capturing the business requirements for the system.

The nature of agile software development states a requirement that user interface and user experience design process should suit the iterative way of working. Depending on the organization, this design work may be the responsibility of a team, a single person, or multiple teams. In all these cases communication is naturally important so that the designers, developers, and all stakeholders are on the same page of the design. Thus, it is a good practice to establish a common understanding by visually capturing a new feature's UI design. This can be done, for instance, by paper and pencil sketches and wireframes. The design may also be further clarified and emphasized by having design sessions where designers, developers and other stakeholders work on the design. Additionally, more advanced, but still relatively quick to produce, prototypes may be developed to better capture the interactions in the user interface. (Brown 2013, pp. 41, 54-55)

One way to approach the UI design of a new feature in a sprint is to make the design one sprint ahead of the actual implementation. Naturally, the aforementioned sketching and prototyping methods can be utilized. This helps having a better vision of the design before he implementation. Then, the design can be demonstrated at the end of the sprint. Another approach is to design the user interface parallel to its development which naturally requires good collaboration between the developers and the UI designer during the sprint. This reduces up-front design work and additionally the risk of wasted effort is lower because issues in the design may be identified and corrected during the implementation work. (Brown 2013, pp. 45-47, 153-154)

The actual implementation of the user interface depends on the platform of the application and the chosen technologies. For example, the UI for an application running on a web browser is implemented by using web technologies, such as HTML (hypertext markup language), JavaScript, and CSS (cascading style sheets). These implementation technologies are presented in more detail later.

User research and usability testing

User research methods are used to gain feedback about the UI design from the users. Some forms of user research are, for instance, usability testing, focus groups, interviews, A/B testing, and surveys (Rohrer 2014). The user research methods improve communication with the users and helps addressing found issues in the design. User research is important because developers and users often have different backgrounds, levels of knowledge, expectations, and attitudes towards the application. Additionally, issues in UI design that are discovered early are naturally also more easier to fix. Therefore, continuous testing and user involvement in the design is important in all phases of development. (Galitz 2002, pp. 702-703)

Usability testing is one of the most powerful user research methods because it emphasizes the user behavior and qualitative information (Rohrer 2014). That is, the results contain information on what the user really does rather than what the user says, and the results often provide answers to questions why user feels certain way and how to address the issues in the design. Galitz (2002, pp. 718-719) defines usability test with some typical characteristics. First, usability test often takes place either in a lab environment or under controlled conditions that resemble actual work environment. Additionally, the test contains tasks that the user performs by using the system. All detected problems, errors, confusion, frustrations, and complaints are recorded and the usability test conductor discusses them with the user.

User research and usability testing can be included to agile development sprints by different ways. For instance, a frequent, regular testing session with the customer is beneficial but naturally requires good scheduling. These sessions may include the users trying out sketches, prototypes, working software, and basically anything that has been produced. The session can be a proper usability test but it may have to have a narrower focus and shorter timeframe. Moreover, the fast pace of the agile sprints are better suited for some user research methods, such as, smaller scale usability tests, interviews, surveys, web analytics, and unmoderated remote usability tests. Larger scale usability tests and user research methods may be performed out of normal sprint cycle and can possibly be tied to achieving certain milestones in development. Nevertheless, in all cases it is important that the recorded feedback is turned into backlog items that really affect the design and development, and reflect the users' needs. The received feedback should also be discussed with the developers so that the user's point of view is better understood. (Brown, pp. 56-63)

UI design can also be assessed by methods that do not necessarily require participation of actual end users. One such method is guidelines review which is performed by inspecting the UI against the design guidelines and standards of the organization (Galitz 2002, p.710). The reviews are mainly done by the developers and UI designers, and typically general issues in the design are detected.

Another method is heuristic evaluation which is performed by a group of user interface design experts. The application and its UI components is compared against a set of usability principles which often are more general rules that describe common properties of a good UI design rather than very specific guidelines. Preferably each evaluator should perform the evaluation independently so that the findings are not affected by other evaluators. The evaluator may have a domain expert as a supporter so that the usability evaluation is not hindered by possible lack of domain specific knowledge. Heuristic evaluation can be performed by using actual working software but also lower level prototypes. (Nielsen 1995)

Role and limitations of test automation in user interface testing

As it has been stated before, UI test automation work best in the role of regression testing, that is, helping to ensure that existing functionality has not been affected by changes to the application. By examining the definitions of user experience and usability, it is evident that those characteristics cannot be assessed by automation. For instance, automation cannot give information on how pleasant a product is to use or how easy it is for a user to learn to use. Some specific aspects of usability and accessibility, such as the application's support for keyboard navigation, may be able to receive some supplemental feedback from automation (Harty 2011) but automation can by no means replace proper user experience and usability testing. Thus, the user experience and usability aspects of the UI design should be tested by other means, such as the methods described earlier in this chapter.

Automated regression tests on UI level do naturally have their own challenges as well. This is especially the case if the tests are directly interacting with the user interface. Such tests are vulnerable to changes in the UI design and may thus cause false positives. Nevertheless, testing on the UI level is important because that is the layer where normal user interaction with the application takes place. A good solution to make the UI tests more maintainable is to make use of the test libraries (Marick 1998), a method that was mentioned earlier. Another name for this method is application driver layer (Humble & Farley 2010, p. 198) and in web environment it may be called page object design pattern (Fowler 2013, SeleniumHQ.org 2016). Nevertheless, the main point of this method is that the test case code does not directly access the UI but rather via a specific API. Thus, changes to UI requires updating the API and not the test cases which should make the tests more maintainable. (Humble & Farley 2010, pp. 192-193, 198)

3.2 Web technologies

Web technologies are used in implementing applications that are based on the cooperation of web browser clients and a web server. The application UI displayed by the browser is based on rendering structured hypertext markup language (HTML) documents and the application functionality can be enriched by variety of other technologies.

Delivering the presented content, functionality, and user experience requires often complex logic and design both in the client (frontend) and the server (backend). Moreover, applications may also make use of web services running on other remote servers. Figure 3.1 presents an illustration of commonly used web technologies. (Casteleyn et al. 2009, pp. 2, 23, 49)

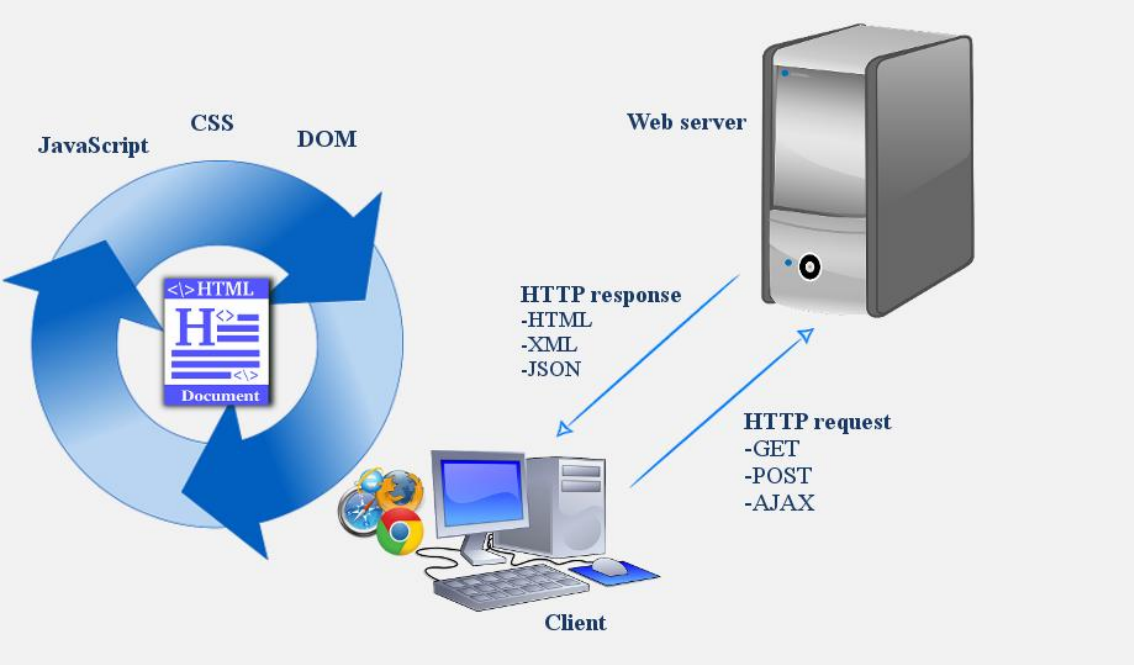

Figure 3.1 *Many modern browser-based applications are systems that have their logic distributed in both server and client. Many technologies are involved in the implementation of this logic and the communication between client and server.*

The technologies presented in Figure 3.1 mostly concern the user interface of the application. The basis of UI technologies in browser based applications are practically always in HTML, JavaScript, and CSS but the server side logic often has a variety of technologies and frameworks to choose from. Such backend technologies include, for instance, PHP (hypertext preprocessor), Ruby on Rails framework (Rubyonrails.org 2016), ASP.NET framework (Asp.net 2016), Node.js (Nodejs.org 2016) JavaScript runtime environment, to name a few, and naturally different database technologies. Although, it should also be noted that the application UI is often also implemented by utilizing different frameworks, for example, Angular.js (Angularjs.org 2016) and React (Facebook.github.io 2016) JavaScript frameworks, and Bootstrap frontend framework (Getbootstrap.com 2016). However, for the purposes of this thesis it is enough to understand how automated user interface tests interact with the application by only considering the basic underlying technologies that are used to form the frontend user interface. Moreover, for the purposes of UI test automation the chosen server side implementation technologies are mostly irrelevant as long as the backend provides the required functionality. It should be noted that automated user interface tests invoke both frontend and backend code and thus issues in both layers may be detected.

Next, some of the main technologies involved in the operation of browser-based application are briefly presented.

The hypertext transfer protocol (HTTP) and HTTP requests

The hypertext transfer protocol is used in specifying requests of resources between a client and a server (Casteleyn et al. 2009, p. 10). A web browser requesting an HTML web page from a web server is a typical example of an HTTP request. This kind of HTTP traffic is visible in Figure 3.1 between the server and the client.

Requested resources are identified by a uniform resource locator (URL) strings. An URL is formed of the used protocol, name or IP address of the server hosting the resource, an optional port number, the path to the resource, and the resource name. The URL can also contain query string parameters to provide additional data or instructions to the request. (Casteleyn et al. 2009, p. 10)

Further, HTTP requests have different HTTP methods available. GET method is used by the client to fetch a resource from the server. On the other hand, POST method is used to make a request that contains input, often more complex than the query strings, to be processed by the server. Thus, the request using POST method can contain attachments, such as files or other structured data, stored in the request body. (Casteleyn et al. 2009, p. 11)

The server sends a response to the client containing the requested resource, provided that such a resource exists and that the user is allowed to access to the resource. Therefore, the response also contains a status code that expresses the result of the request, "200 OK" or "404 Not found", for example. Further, additional information may also be passed between the client and the server by specified header fields in the requests and responses. (Casteleyn et al. 2009, p. 11)

The hypertext markup language (HTML)

The hypertext markup language is used in HTML documents to define the content and visual formatting of web pages. A web page, in other words, an HTML document processed and rendered by a web browser, consists of HTML elements. The beginning and end of each element is marked by tags, and the content of the element is located between these two tags. In some cases the element may not have any content and thus does not require an ending tag. In Figure 3.1 the HTML document can be seen as the center of the web page that is augmented by other technologies. (Casteleyn et al. 2009, p. 11) It should also be noted that different browsers and client devices have their own unique ways to interpret and present the HTML structure of the application UI (Casteleyn et al. 2009, pp. 115-116).

An HTML element may also have a set of attributes in the start tag. These attributes are name-value pairs that are used in defining additional properties to the element (Casteleyn et al. 2009, p. 11). Some attributes are common and can be used with all elements but some are specific only to certain elements (WebPlatform.org 2015).

Cascading style sheets (CSS)

Cascading style sheets is a language that can be used to specify how documents are presented (WebPlatform.org 2014). CSS allows the separation of the presentation from the content of the document. Further, multiple web pages may also use the same CSS file to achieve a common presentation style between the pages (Casteleyn et al. 2009, p. 14; WebPlatform.org 2014)

In practice, CSS is based on a set of rules defined by the web designer. Moreover, these rules are interpreted by a web browser to present the document. Each rule contains two parts: a selector and a style declaration. The selector defines which elements the style declaration will apply to. The elements can be selected by their name, by a value of the element's attribute, and by more other types of selectors or their combinations. On the other hand, the style declaration contains style property-value pairs to affect the presentation of the selected elements. The property defines a style characteristic to be affected and the value defines the actual effect. The property "color" and value "blue", for example, define that the color of the text in the selected elements is blue. (Casteleyn et al. 2009, p. 14; WebPlatform.org 2014)

The document object model (DOM)

The document object model is "a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents" (W3C 2009). Essentially, DOM represents a web page as an object, a document object. In DOM, the document is presented a collection hierarchical nodes, each node representing elements, element attributes, or text content of the document. Further, each of these nodes is also an object with its own set of properties and methods. (Keith & Sambells 2010, pp. 31-41)

In practice, the document object can be used in browser client scripts to locate and access the nodes on the web page. For example, the contents of the element nodes and the values of the attribute nodes can be read and modified. Further, new nodes can be created and added as part of the document structure. Similarly, existing nodes can be removed. Moreover, any modifications made to the document object and its nodes are dynamically updated to the web page in the browser. (Keith & Sambells 2010, pp. 100-103)

Asynchronous JavaScript and XML (AJAX)

AJAX is a concept in web development that allows loading additional content from web server without having to load a whole HTML page again. AJAX utilizes client side scripting, written in JavaScript, to make HTTP requests to the server. Naturally, the client side script also has to handle the response and use the DOM to make the content from the response available to the user. Moreover, the request is made asynchronously which effectively means that the user can continue interaction with the web page while the server is processing the request. (Casteleyn et al. 2009, p. 26; Keith & Sambells 2010, p. 116)

AJAX makes use of XMLHttpRequest object to request data from the server (Keith & Sambells 2010, p. 116). In spite of its name, XMLHttpRequest supports any text based format for the data exchange between the client and the server, including extensible markup language (XML) (W3C 2014). Another often used data format in XMLHttpRequest is JavaScript object notation (JSON).

Representational state transfer (REST)

Representational state transfer is practically not a web technology but rather an architectural style used in web services. Especially, applications using AJAX often interact with a backend API that is based on REST principles. First, operations using the API are based on standard HTTP methods that are used as intended, for instance, GET to retrieve a resource and PUT to update a resource. The resources are identified by URLs. The representations of the resources are sent as data, such as XML or JSON, and the type of the data can be identified by utilizing media types, also known as MIME types. Additionally, the server does not save state of the clients but rather the requests should contain all state information. (Casteleyn et al. 2009, pp. 53-54)

3.3 Practical methods in web user interface test automation

Implementing tests that interact with a web user interface have to consider some aspects. First, the interaction with the browser itself has to be automated. Additionally, the test cases can use different layers of abstraction in order to achieve good maintainability and readability. Now these aspects are briefly discussed.

Controlling the web user interface

Web user interface test automation often makes use of WebDriver API (W3C 2015) which provides an interface for remote control of browsers. One common tool for that purpose is Selenium (SeleniumHQ.org 2016). In Selenium, the user interface elements in a web page are located via the API based on their inner HTML structure. In practice, this means that the HTML structure of the web page, such as element tags or attributes and their values, has to be known in order to access and interact with the elements. The

API offers methods for actions, such as clicking and typing input to the elements (Selenium.googlecode.com 2016). Selenium is discussed in more detail in chapter 4 because it is also utilized in M-Files UI test automation.

Another approach is to use visual image recognition in the interaction with the user interface. For instance, SikuliX (SikuliX.com 2016) is a tool that utilizes this method. Additionally, the tool can interact with the recognized UI elements by automatically controlling mouse and keyboard. This approach may be considered, for instance, if the internal structure of the UI is not known. This type of recognition can also be utilized in verifying the correct display of visual content on the web pages (SikuliX 2016). Such verification can be very difficult to achieve by only reading the HTML structure of the elements because the errors in the visual display are more likely to be in the CSS.

Structure of the user interface tests

One method in user interface testing is to implement the tests by using a domain-specific language (DSL). These languages attempt to capture concepts of a particular domain and they can be divided into internal and external domain-specific languages. Internal domain-specific languages are effectively expressed by utilizing code style practices that make the code more understandable by readers that are familiar with that particular domain. That code may thus be better understood by customers, other more business oriented stakeholders, and naturally, the test developers. In practice, an internal DSL is not a separate language but rather a specific implementation written by using basically any programming language which can be called host language. An internal domain-specific language can be used in test cases to separate the actions in the domain from their implementations. Effectively, such test cases focus on the familiar concepts in the domain and not how the actions are carried out in detail. Internal DSLs can utilize all the features of the underlying host language while still often retaining a fair level of understandability. (Fowler 2008, Humble & Farley 2010 pp. 198-201)

On the other hand, external domain-specific languages are separate languages that are required to be parsed in order to be executed (Fowler 2008, Humble & Farley 2010, p 198). Such languages can be defined, for instance, by using Cucumber tool (Cucumber.io 2016). In practice, Cucumber allows writing tests in plain English, or other supported languages, augmented with specific keywords. Such keywords are, for example, "Given", "When", "Then", and "And". By using these keywords and adding a row of text after them, test steps are formed. For instance, a very simple test case could be formed by using following steps:

- Given I have 100 Euros
- And my bank account balance is 0 Euros
- When I deposit 80 Euros to bank
- Then I have 20 Euros remaining

- And my bank account balance has changed into 80 Euros

Such test cases are very understandable but, naturally, the steps have to be defined by using a programming language. The matching method defining the test step is recognized by a pattern in the implementing code that is formed of the keyword and the text part of the step (Cucumber.io 2016). These methods effectively form the implementation layer of the tests. This implementation layer, in turn, may call another layer of code that actually performs the required operations in the step based on the implementation details of those operations (Humble & Farley p. 197). That next layer of code, called application driver layer by Humble and Farley, can actually be an internal DSL. Nevertheless, test cases that are defined by using an external language on the top layer can be easily discussed with the customer and basically all stakeholders are able to collaborate on them (Humble & Farley 197-198).

Another point of view to the development of automated web user interface tests is to separate the implementation of the UI from the test cases. Thus, there should be a specific layer of code that knows how to interact with the inner HTML structure of the web pages in the application. This method increases the maintainability of the UI tests because changes in the UI require modifications in this specific layer of code and not in the tests themselves (Humble & Farley 2010, pp. 201-204). This layer can be implemented, for instance, by using Selenium WebDriver API. Moreover, this UI separation layer should be divided into different components so that they each represent individual user interface structures of the web pages in the application. This solution emphasizes how the UI is viewed by the user and not how it is internally implemented (Fowler 2013). Additionally, operations using a text field, for instance, should use string parameters in the methods for input and output operations and, similarly, checkboxes should use boolean values in order to make the layer easier to use. In practice, this layer simply encapsulates the user interface which provides benefits to both the maintainability and readability of the test cases (Fowler 2013). The use of this UI separation layer method can be seen in Figure 3.2.

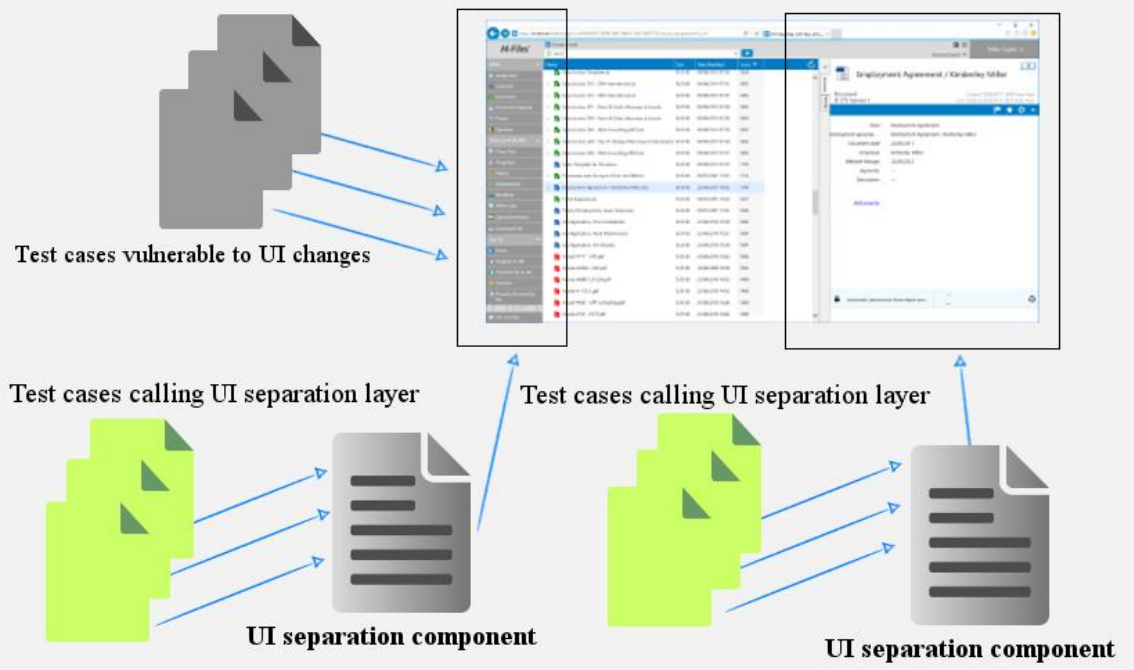

Figure 3.2 *UI separation layer encapsulates the user interface so that knowledge of its inner HTML structure is not required. The components in the layer represent parts of the UI and they increase clarity of test cases. Additionally, only the components need to be modified if the UI changes.*

This code layer that separates the UI from the tests can naturally be utilized with domain-specific languages. Effectively, that layer can be regarded as an internal DSL (Humble & Farley p. 198). However, the UI separation layer can also be implemented under an internal domain-specific language if that DSL focuses purely on business requirements. On the other hand, external DSLs can similarly be layered on top of the UI separation layer. Effectively, by using the aforementioned Cucumber example, calls to the UI separation layer may be done in the test implementation layer or alternatively under it in the application driver layer. Again, this basically depends on how many abstraction layers the test architecture contains.

In addition to the abstraction layers, also the starting point of the test cases can be considered. For instance, McMahon (2009) describes a tree-like design and a web-like design for UI tests. The tree-like design is an approach where all the tests begin from the same starting point and then gradually branch to follow different paths in the system. According to McMahon, the problem with this design is that any failure along the path causes tests to fail even if they were supposed to test another feature further along the path. On the other hand, the web-like design presents multiple starting points in the system. This approach may make failure analysis easier because a failure should only affect a smaller number of test cases. Also, the length of a test case is a factor when analyzing a failure. A short test is often easier to analyze and thus larger tests can be refactored into smaller cases (McMahon 2009).

4. USER INTERFACE TEST AUTOMATION OF M-FILES WEB

This thesis considers the user interface test automation tool of M-Files Web application and its utilization in the product development process. Thus, M-Files organization and the product development process are examined in order to provide an overview of the context for this thesis. Then, the user interface test automation tool and the related test automation process is discussed in detail. Finally, the identified points of improvement in the UI test automation are described.

4.1 M-Files product and M-Files Web

M-Files is an enterprise content management system developed by M-Files Corporation software company. M-Files offers solutions to many industries, including government, manufacturing, healthcare, and logistics. M-Files can thus be utilized, for instance, in managing contracts, purchase orders, invoices, and standard operating procedures, and also in controlling business workflows and processes. The system manages information based on content metadata, that is, the right information is found based on what it is rather than where it is. M-Files offers several features, such as powerful searches, dynamic views, version history, workflows, permission control, electronic signatures, and integration with Windows Office products. M-Files has clients for different platforms: M-Files Desktop for Windows, M-Files Mobile for iOS, Android, and Windows Phone, and finally, M-Files Web for access via web browsers. M-Files Desktop integrates directly with Windows Explorer interface and effectively acts like a normal drive location for the user. Moreover, M-Files offers similar UI across all platforms. M-Files Desktop user interface can be seen in Figure 4.1. (M-Files.com 2016)

In M-Files, the content is stored as objects. For example, a document object consists of a document file and the metadata associated with it. However, objects can also be created without any files, as pure metadata. Such objects may be used, for instance, managing a customer or project database. The metadata associated with objects are called property definitions. Each property definition has a data type, such as text, integer, or the property value can be chosen from a list of predetermined values. (M-Files.com 2016)

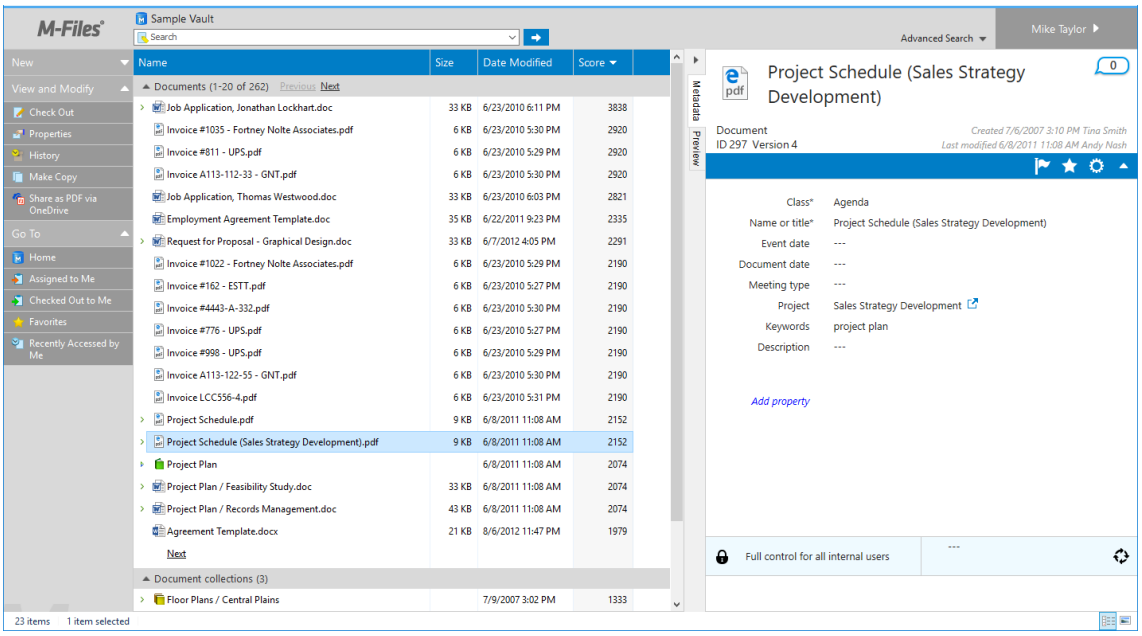

Figure 4.1 *M-Files Desktop user interface contains task area on the left, listing area in the middle, and right pane with the metadata card of the selected object. Search functions are located above the listing area in the middle.*

The objects are stored in M-Files vaults and the users can access these vaults by connecting to M-Files server via the aforementioned clients. Additionally, administrator users can configure vault metadata structures, other vault settings, and M-Files server settings via M-Files Admin tool. Moreover, M-Files can be set up as a on-premises installation, in cloud, or as a hybrid solution. (M-Files.com 2016)

M-Files also includes an API, simply called M-Files API, a collection of classes and operation interfaces that can be used with Visual Basic, VBScript, C++, and all .NET languages such as C#. Both client operations and server operations can be performed by using this API, such as creating and modifying objects, making searches, and modifying vault metadata structure. Effectively, most operations that are available by using the clients programs are also available via the API. Additionally, M-Files offers a REST-like API called M-Files Web Service. This API has a more limited range of features but supports reading and modifying objects, and reading document vault structures. The functionality of all M-Files clients make use these two APIs but they are also available for use to M-Files users. (M-Files.com 2016)

M-Files Web is an application that is used via a web browser. The application is implemented with standard web technologies, such as HTML, JavaScript and CSS, and thus it supports various browsers. M-Files Web application and M-Files Web Service are both deployed on Internet Information Services (IIS) web server. By default configuration, the web server should on the same computer where M-Files server is installed but it can also be located on a separate computer. (M-Files.com 2016)

User logs in to M-Files Web via a login page and then the contents of a vault can be accessed by making searches or by navigating to views via the home page. The UI of this page is composed of task area, listing area, right pane, and search. These components are visible in Figure 4.2.

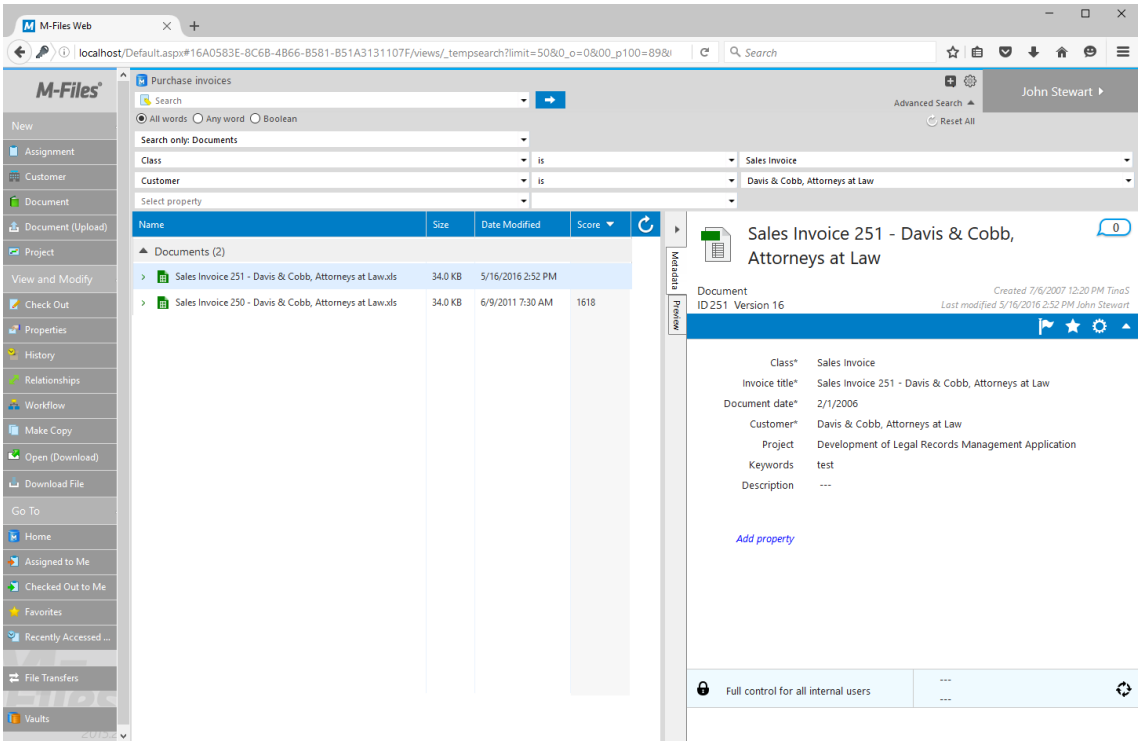

Figure 4.2 *User interface of M-Files Web is very similar to the user interface of M-Files Desktop. Advanced search is shown expanded in top middle part of the UI.*

The task area on the left contains options for various operations, such as, creating new objects, checking out an object, viewing object version history. The task area also contains quick navigation to useful views, such as the "Favorites" and "Recently accessed by me". On the other hand, the listing area in the center contains objects when they are located by searches or by navigating to views. In addition to objects, the listing area may contain views, sub views inside other views, and virtual folders that can group objects in a view based on metadata. The right pane contains the metadata card of a selected object. The metadata card can be used for viewing and modifying the metadata of a selected object. Additionally, the right pane contains a preview tab for previewing the selected document. Finally, the search functions are located on the top of the UI. The search bar allows the user to perform a quick search to look for objects that contain the search word in their metadata, file contents, or both. User can also enter advanced search options to define criteria for searched objects, for instance, the object must have a certain value in a specific property. These aforementioned four main UI components are by default visible in most situations, only the content in them changes. It should be noted that authenticated administrator users can also access a configuration page and

modify, for instance, language settings and UI layouts of M-Files Web. (M-Files.com 2016)

4.2 Product development process and testing in M-Files

New M-Files product version releases are managed by milestones. Each milestone has certain criteria that have to be fulfilled. Currently, new major versions for M-Files are released approximately every 24 months and minor versions every 6 months. Additionally, service releases are released approximately every 3 months. Major M-Files version release cycle is composed of multiple development phases: release planning, feasibility studies and UI concepts, implementation and testing, and finally, finalization, system testing and releasing. Each development phase has at least one milestone and the phases may overlap to some extent. (M-Files Corporation 2016a)

Basically all M-Files client products – that is, Desktop, Web, and Mobile – follow the same general direction concerning their functionality. Usually, new features are first implemented in M-Files Desktop whose functionality, and look and feel is then used as a benchmark for other platforms. Further, M-Files Web does not have its own release cycle and is tied to the release cycle of M-Files Desktop and server software. Therefore, the design process of M-Files Web cannot be regarded as an isolated process.

M-Files product development process is influenced by agile Scrum method. Each development team has a prioritized backlog of working items that are called user stories. The development work of a team is based on two-week long sprints that consist of several events. Thus, each phase in the release cycle consists of multiple sprints which can be seen in Figure 4.3. Before a sprint, a prioritization meeting is held to decide the order of the user stories and to discuss possible design and development changes. At the start of the sprint, a planning meeting is held in order to decide which user stories the team can implement during the sprint. At the end of the sprint, a review meeting is held where each user story is checked against definition of done and demonstrated. Finally, the user story is accepted by the product owner if no deviations are found. (M-Files Corporation 2016a)

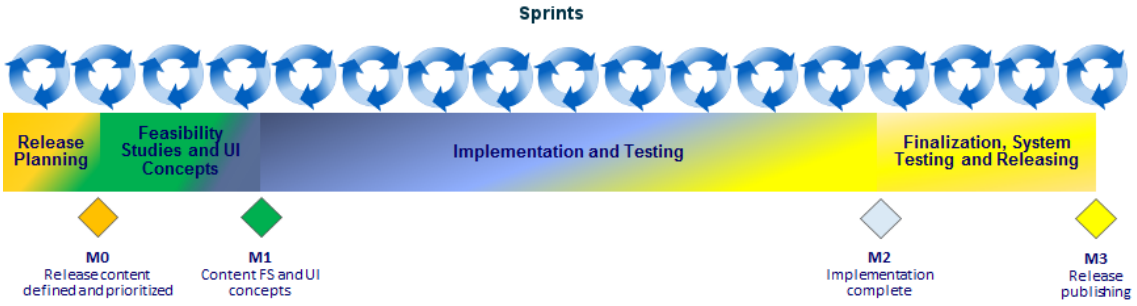

Figure 4.3 *M-Files major version release cycle with its phases and major milestones. Development work is done inside the phases in sprints according to an adaptation of Scrum method.*

Definition of done serves as a major quality assurance control point for the user stories. For instance, it sets certain criteria for user interface style and required testing. Thus, a user story can only be accepted if the user interface meets UI style requirements, and testing of the user story has been done by someone else than the implementing developer. Additionally, the test cases and their execution must have been appropriately documented and found issues reported to the issue tracking system. (M-Files Corporation 2016b)

User story testing is mainly done by M-Files testing team. In some rare cases, a developer can test another developer's user story. In addition to already mentioned system testing and user story testing, the testing team also performs performance and security testing. Additionally, manual smoke testing on all release builds is done by the testing team. Moreover, manual system testing is performed before major releases.

The testing activities and test results are managed by using a specific M-Files vault called Test management vault. This vault has a custom metadata structure that supports especially manual testing in the organization. The vault has a workflow for the testing of new user stories which emphasizes visibility to the current state of testing. Additionally, test cases that are executed for the user stories are also documented as objects in the vault.

Issue tracking is also managed by an M-Files vault called Tracker. Defects and improvements are created as issue objects. All details of the issues are filled as metadata which means that they are easily located by searches or in specific views. The issues contain the steps required to reproduce the issue, expected and actual results, and version where the issue was detected. Also, for regression defects the version where the issue was known to previously work correctly may be filled. The issues are assigned a priority and they are scheduled for a targeted release. (M-Files Corporation 2016c)

The testing team also develops and maintains three different types of automated regression tests. First, automated integration tests are developed for M-Files API by using NUnit framework (NUnit.org 2016). Second, automated UI tests are developed for M-Files Desktop and M-Files Admin by using TestComplete Platform (SmartBear.com 2016). Finally, automated UI tests are also developed for M-Files Web. The solution and used technologies for UI test automation of M-Files Web are described in detail later.

Currently, the automated NUnit tests are executed for all new M-Files builds. The testing builds are controlled by Teamcity CI server that was already briefly presented in chapter 2.2. A testing build is started as soon as a new M-Files build is available. The build consists of several steps, such as downloading the M-Files installer from builder computer, installing M-Files software, setting M-Files product license, compiling the tests, and running the tests. Most of the setup operations are done by PowerShell scripts. The results of the tests can be viewed in Teamcity's web dashboard. If there are some new failing tests in the build, the results are also sent as email notifications to the developers who have made commits towards the build. On the other hand, the automated UI tests for M-Files Desktop and M-Files Web are executed about every other week and more often when a release is closer. The process for UI test automation is more manual and does not utilize CI server software. Next, the M-Files Web UI test automation tool and testing process related to it is discussed in detail.

4.3 Current state of user interface test automation

Web user interface test automation for M-Files Web is a tool developed and maintained by M-Files testing team. Currently, over 1700 automated UI test cases have been developed by using this tool. This test automation tool is used for regression testing and smoke testing of M-Files Web.

4.3.1 Technologies used in the user interface test automation tool

The current implementation of M-Files web user interface test automation tool is written in Java programming language. Moreover, the test automation tool utilizes TestNG testing framework in the implementation of the test suites and their test cases. Additionally, the dependencies of the test automation project are controlled by Maven software project management tool (Apache.org 2016b). Further, the project makes use of Selenium WebDriver API (SeleniumHQ.org 2016) to automate the controlling of web browsers and thus the web application and its user interface. Also, Selenium provides a mechanism for the system to run tests on multiple remote computers and browsers in parallel. Next, this chapter briefly introduces all the aforementioned tools of the UI test automation of M-Files Web.

TestNG framework

The following description of TestNG framework is adopted from the framework's official documentation (TestNG.org 2015). TestNG is a testing framework for Java programming language. Further, the framework makes use of TestNG annotations that are inserted to the test code. On the other hand, an XML file is used to configure information about the tests that will be run.

Each test suite in TestNG is represented by an XML file. A test suite may contain one or more tests which in turn consist of one or more test classes. Further, each test class can be defined to include certain test methods. Additionally, custom parameters can be defined on suite level or test level. Test level parameters have precedence over suite level parameters if both parameters have the same name. Thus, test level parameters can be used individually to override general suite level parameters if required. (TestNG.org 2015)

TestNG also provides a mechanism for parallel execution of tests. The parallelism is defined in the XML file by using the "parallel" and "thread-count" attributes. These attributes can be used both on suite level and test level. The value of the "thread-count" attribute specifies the number of parallel threads in execution. Further, the value of the "parallel" attribute can be either "tests", "classes", or "methods" if specified in suite level. However, only "classes" and "methods" are valid values in test level. (TestNG.org 2015)

TestNG test classes are Java classes that use TestNG annotations. Further, each annotation has its own attributes to provide additional information for test configuration. The test classes contain individual test methods that are marked by the @Test annotation. Some examples of useful attributes for @Test annotation are dataProvider, dataProviderClass, groups, and timeOut. Additionally, TestNG has useful annotations for defining methods to initialize the test environment or bring the environment back to its original state after each test. Such annotations are, for example, @BeforeSuite, @BeforeClass, @AfterClass, @BeforeMethod, and @AfterMethod. (TestNG.org 2015)

The dataProviderClass and dataProvider attributes of @Test annotation are used to define a class and its method to provide data to the test. The data provider method provides the test with one or multiple arrays of Java objects to be used as test data. The test is executed as many times as the number of arrays produced by the data provider method. Thus, the same test can be executed multiple times but with different test data. (TestNG.org 2015)

Test cases can be marked to belong to certain groups by using the "group" attribute of the @Test annotation. The group names are specified as the values of the attribute. Further, these groups can be referred to in the TestNG configuration XML file in suite or test level. The referred groups can either be included or excluded from the test suite.

Thus, the grouping allows the user to further structure the tests inside a test class or make tests belong to a same group between multiple classes. Also, the user can with relatively ease run different sets of tests by only modifying the XML configuration. (TestNG.org 2015)

Maven

Maven is a software project management and comprehension tool. It especially aims to provide means to manage project dependencies, to make the build process easier, and to encourage the use of standard conventions and practices in the software development process. (Apache.org 2016b)

Maven build process is based on defined build lifecycles. Further, the build lifecycles consist of build phases. Maven has a built-in build lifecycle called "default" which contains many individual phases, for example, validate, compile, test, package, integration-test, verify, install, and deploy. Moreover, each of these phases in turn are composed of Maven plugin goals. A single goal can be run in one or multiple phases of the build. Thus, the build process can be configured by binding different goals and their combinations to the build phases. Maven can be instructed to run a certain phase in a build lifecycle but that also causes all previous phases to be run. (Apache.org 2016b)

Maven makes use of a specific XML file called pom.xml (POM, Project Object Model) to manage the information and build configuration of the software project. Many default configurations are already defined in the so called Super POM that all user defined POMs extend. For example, the Super POM contains the default directory structure for the project, default repository information for project dependencies, and default Maven plugin information. The actual POM file of the project can be used to define dependencies to third party components and also to configure Maven plugins. (Apache.org 2016b)

Maven has a central repository hosted by Sonatype Inc (Sonatype.org 2016) which contains many open source libraries and components published by open source organizations and individual open source projects. Further, components defined as dependencies in POM are automatically downloaded from the repository if they are not already present (Apache.org 2016b).

Selenium

Selenium is a set of tools for different purposes to support test automation of web applications. First tool in the set is Selenium WebDriver, a programming interface that can be used to directly control a web browser. The current latest version of Selenium WebDriver provides an object-oriented API that supports the automation of most major browsers. Second, Selenium Grid is a tool designed for parallel execution of tests on a distributed set up of different computers or virtual machines. Selenium Grid consists of

a hub and nodes that are registered to the hub. Third, Selenium IDE (Integrated Development Environment) is a tool for recording reusable test scripts. Selenium IDE is a Firefox browser plugin intended for prototyping rather than full-fledged test automation development. (SeleniumHQ.org 2016)

Developing tests with Selenium WebDriver usually revolves around utilizing the methods of the WebDriver interface and the WebElement interface. WebDriver interface provides methods for controlling the browser and selecting the elements in the web page. Elements can be located by different means, for instance, by element's tag name, by element id attribute, or by CSS selectors. Located elements are objects that implement the WebElement interface. (Selenium.googlecode.com 2016)

WebElement interface provides methods for simulating user actions on the web page elements, such as clicking and typing. Additionally, elements' content and attributes can be accessed by using the interface. Moreover, the other elements contained in an element can also be accessed using methods similar to the ones specified in WebDriver interface. (Selenium.googlecode.com 2016)

Selenium Grid allows parallel execution of tests in a set up of different testing environments. In practice, a selenium hub process is running on a computer and answers in a specific port. Further, a node is registered to the hub by a command that specifies the node's configuration. The configuration is composed of information about the computer where the node process is running. Thus, the configuration contains information such as the operating system, different browsers available in the node, and the maximum number of allowed concurrent instances of those browsers. Both the hub and nodes are started by running the Selenium server standalone executable with specific command line parameters, including whether Selenium takes the role of a node or a hub. Additionally, some browsers, such as Internet Explorer (IE) and Chrome, require additional driver executables on the node computer. (SeleniumHQ.org 2016)

Tests using Selenium WebDriver can request the Selenium Grid for a node with a specific configuration of operating system, browser, and browser version. In test source code this configuration is expressed by a set of capabilities. These capabilities are passed to the constructor of a RemoteWebDriver object along with the URL of the Selenium hub. Moreover, the hub will assign the test to a node with the requested capabilities and starts the browser. On the other hand, the test will fail if none of the nodes match all the requested capabilities. (SeleniumHQ 2016)

4.3.2 User interface test automation implementation

Web user interface test automation tool of M-Files Web is developed as a Java project managed by Maven. The project is entirely focused on automated web testing so the Maven configuration is not used for building any M-Files applications. All dependen-

cies to third party components, such as TestNG and Selenium, are expressed in the POM of the project. Test source code and other related files are stored in the company SVN repository but the third party components are fetched from the Maven central repository.

Test source code is divided into two packages: MFClient (M-Files Web client) and generic library. Further, MFClient is divided into three packages: pages, wrappers, and tests (test classes). Pages and wrappers are Java classes implementing the UI separation layer, or in other words, page object design pattern for M-Files Web. Thus, the pages and wrappers may be regarded as implementing an internal domain-specific language. Further, the pages and wrappers implement methods for interacting with the application on its web UI level. On the other hand, the test classes are TestNG classes, each representing a certain area in the functionality of the application. Finally, the project also depends on resource files for test data and a couple of console applications. These latter mentioned files also have their own place in the project structure and will be discussed later in this chapter.

A test suite for web test automation is configured using a TestNG XML file. The configuration file contains test elements that each usually contain a single test class. Each of these tests can be configured to be run in parallel. Further, the tool requires several parameters to be defined either on suite level or test level. Some important suite level parameters are, for instance, login page URL and configuration page URL of the AUT, URL and port of Selenium hub, used browser, and M-Files username and password to be used in tests. Test level parameters define the name of the M-Files vault to be used in the test and the name of the test data Excel file. However, each of these parameters can be defined on either the suite or on test level. For example, all the tests can be configured to use the same M-Files vault on the suite level, and any test can be defined to use a specific browser on test level. The different technologies and the general flow of web user interface test automation is shown in Figure 4.4.

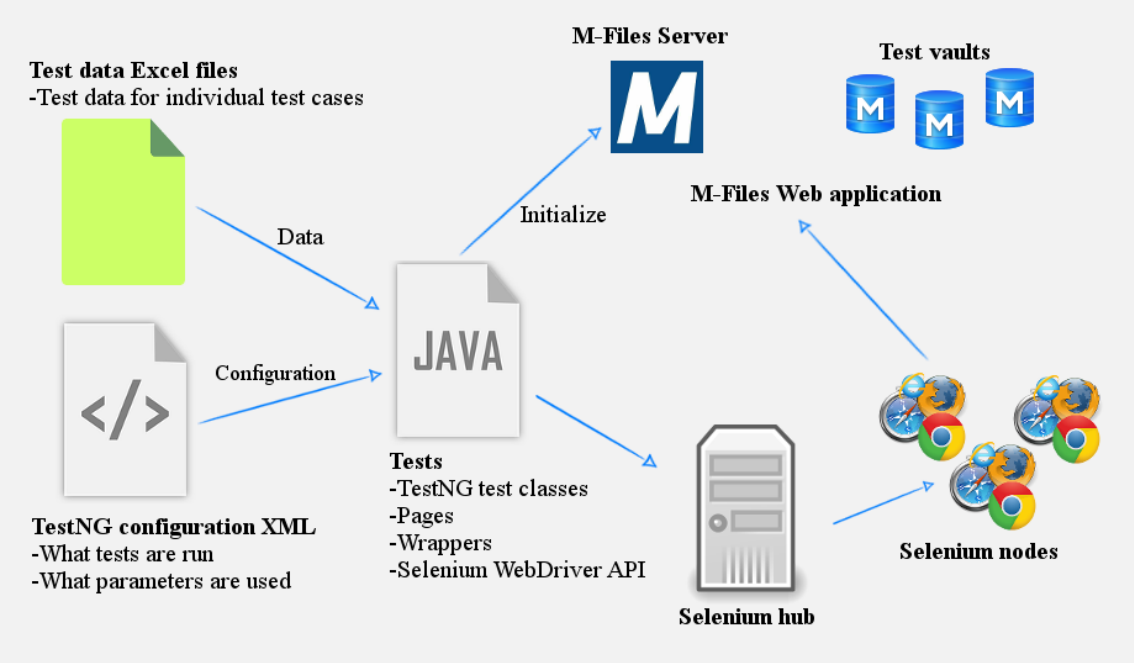

Figure 4.4 *Web UI test automation of M-Files Web application is implemented using TestNG framework and Selenium. Test data is in Excel files.*

Each test class implement a method to be run before the test suite and another method to be run before the test class. The BeforeSuite method runs a console application to make a backup of an existing M-Files sample vault. Moreover, the BeforeClass method runs another console application to restore a vault from the backup file and give it a name according to the value from the parameter in the configuration XML. Additionally, some of the parameters are read from the configuration and stored to variables for further use by the tests. Also, if the tests require specific users to be created to the M-Files server and vaults, those are created before starting the test execution. The console applications that communicate with M-Files API are written in C#.

The tool presents M-Files Web as three different page objects: LoginPage, ConfigurationPage, and HomePage. Each of these objects provide methods for operations in the page and access to page content. Further, the pages provide access to wrapper objects that they contain and thus providing even more operations. HomePage object represents M-Files Web application once the user has logged in. Thus, the HomePage object provides access to a wide range of operations used in test cases.

The wrappers are Java classes that represent different UI components in M-Files Web. Some of these components are integrated parts of the page and thus can be accessed via the page objects. However, some wrappers are constructed dynamically inside the test cases, page objects, or other wrappers when that specific UI component is interacted with. For example, SearchPanel is wrapper that represents the UI component in the home page that provides the user with quick search and advanced search functionality.

On the other hand, MetadataCard is a wrapper that provides access to operations and contents of M-Files metadata card.

Practically every test case has some test data specified to it. This test data is stored in Excel resource files. Further, each test class has its own file where each test case has its own sheet. Moreover, the first row of the sheet contains the headers for the test data columns. Thus, each row after the headers specify a set of test data to be used by the test case. TestNG data provider mechanism is used to fetch the data for each test case individually. Therefore, a test is executed as many times as there are rows of test data after the header row.

The general flow of the tests starts with constructing a Selenium WebDriver object. Thus, the test case requests a component in the generic library to provide a web driver. The required browser type, and other configuration information, is identified from the TestNG XML file. Then, a request with corresponding capabilities is issued to Selenium hub. The hub will choose a suitable node and the browser is started in the node host computer. Finally, the browser is ready to be controlled by the web driver object in the test case.

The first user action in a test case is a login operation to the M-Files Web application and choosing the test class specific vault from the vault list. After that, the use case specified by the test case is executed in the vault. Operations in the application are mostly performed by using the wrappers and page objects. Finally, the result of the test case is verified and the browser is closed in the end of the test.

Many automated test cases of M-Files Web focus on operations on documents and objects in an M-Files vault. The used default login operation brings the user to the home page of M-Files Web. The documents and objects, however, dwell in the views and results listings of user-made searches. Thus, many test cases start with a search operation or navigation to a view to access specific target objects. The search parameters, views, and target objects are all defined in the test data Excel files. Therefore, a single test case can be configured to deal with different M-Files object types or to access views with different properties.

4.3.3 User interface test automation process

Currently, UI test automation for M-Files Web is utilized in two different ways: smoke testing and full regression testing. The smoke test set is a short TestNG test class that has test cases for very basic functionality of M-Files Web. This test set is executed frequently for internal development builds that are received from developers. These internal builds are not yet at that point integrated with the code in version control.

On the other hand, a full regression test set is executed on fully integrated builds about twice a month and more frequently when the release cycle is in a later phase. The full regression test round for M-Files Web consists of running all tests for all browsers supported by M-Files: Chrome, Firefox, Internet Explorer (IE) versions 9-11, and seldom for Safari on OS X. The tests are executed for each browser separately, that is, each round of tests has a customized TestNG XML configuration file with a parameter defining the used browser. In other words, a full test round consists of several separate browser specific test rounds.

These browser specific test rounds are run in parallel by using six computers that each have a Selenium hub and a Selenium node. That is, each computer's Selenium hub has a single registered node that is running on the same computer as the hub. Additionally, M-Files server is installed on each of the computers and the tests are run in Eclipse (Eclipse.org 2016) with a TestNG plugin. Therefore, the browser specific test rounds are independent from each other. Moreover, failed tests are executed again as manual test cases and, in case of the manual test passing, as separate automated tests. This step is done in order to identify potential regression defects in the application or problems in the tests.

In practice, executing a test round for an M-Files build is composed of the following steps:

1. Copy M-Files installer to the computer
2. Install M-Files
3. Configure M-Files Web to IIS by using M-Files Admin
4. Start Selenium hub and node
5. Checkout web user interface test code and resources from SVN
6. Modify TestNG xml file with correct run specific parameters, such as M-Files build number
7. Run the TestNG test suite in Eclipse
8. Analyze the results and manually repeat failed test cases. Manual test failures are reported as defects
9. Repeat automated test cases that failed in automation but passed as manual test cases. Failed automated test cases in this step are likely to be false positives that require maintenance and they are listed down.

It should be noted that the aforementioned steps are repeated for each separate browser that is tested in the full testing round. The execution of separate browser specific rounds in step 7 can be done in parallel when all rounds have reached step 7 but all other steps require manual effort.

Three types of test results are produced from the UI test automation rounds: reported defects to the issue tracking system, a full test report of the test automation round, and TestNG result HTML pages of each separate TestNG test suite. The defect reports are created and treated by the same process as any reported issue in the issue tracking sys-

tem. On the other hand, the full report is an Excel file that contains several sheets of data highlighting different metrics, such as pass and failure rates by browser, and causes for failures. The TestNG result HTML pages are produced by the framework of each test suite. The pages contain detailed information on each test case, for instance, pass status, duration, Selenium hub and node details, data provider parameters, log output, and error stack in the case of failure.

4.3.4 Identified points of improvement

This thesis originates from the identified points of improvement in the test automation of M-Files Web that were already briefly presented in the introduction of this thesis. These points of improvement are:

- A full testing round has too long duration in calendar time
- Many test automation related tasks are done manually
- Lack of systematic monitoring of testing duration
- Lack of visibility to the test results
- Test quality issues

The first two issues are regarded with more higher priority and the initial identification of those issues was not performed in the context of this thesis. The identification process began from received feedback from the development teams. The UI test automation was regarded delivering feedback too slowly, and it was found difficult and time consuming to pinpoint which build had introduced specific regression defects to the system. This was discussed with the team members who are working with UI test automation and the amount of manual work required for each test automation round was seen as the main factor preventing the running of the tests more often. The situation was additionally looked into when an internal audit was performed on the parts of the organization where UI test automation is developed. During the audit it was concluded, as according to the earlier observations, that the long duration of testing rounds and the amount of manual work are the main points to be improved in the UI test automation. This was formalized into the research and development department's annual plan as a goal to optimize UI test automation for shorter test cycles. The plan was approved both in research and development department's management group meeting and in the M-Files management group meeting. Test round duration is naturally affected by the manual tasks but in this thesis they are listed as separate issues. This separation is done because some solutions may reduce testing duration without affecting the manual tasks. Additionally, the organization's annual plan included a target to develop methods for performance testing. The third issue, that is, lack of systematic monitoring of testing duration, was therefore also regarded as a point of improvement in UI test automation.

The other two issues were identified during the course of this thesis. The identification was made mainly by observing the current UI test automation process and comparing it

to the principles of continuous integration and good test automation practices, and by discussing these matters within the testing team. First, lack of visibility to test results was identified as a separate issue because good visibility to the state of the build is one of the key elements in CI and better visibility has also otherwise great untapped potential in improving quality assurance practices. Finally, issues in test quality was also regarded as a significant issue because it has also other consequences than long testing duration in the form of failure analysis.

Now all these issues are discussed in more detail in order to provide a more clear understanding of their causes and ramifications. With that understanding, improvement suggestions to solve these issues can be formed in chapter 5.

A full round of tests has too long duration in calendar time

The calendar time duration of a full UI test automation round for M-Files Web is several days. This duration is the time that elapses from the moment when the test automation related activities are started to the moment when the analyzed test results are reported. However, a full round of tests also contains the reporting of the found defects to the issue tracking system. Thus, some results are already available before the final results are reported.

Nevertheless, the overall success of a build becomes evident only after several days of its availability. This long duration also prevents the possibility to run the tests more frequently and thus most builds do not receive automated UI regression testing at all. As per CI principles, it would be beneficial to the developers if any found defect was mapped to the build where it was first introduced to the system. Additionally, with the current process the defect may reside in the system for several days before a full round of tests is run to detect it.

One reason for the long duration is that the tests are run for all browsers supported by M-Files. This approach ensures good test coverage for all supported platforms but at the expense of faster error detection. Both the browser coverage and the fast detection of defects are valued traits in the UI test automation but now the process overly favors the former. Therefore, a solution to solve this tradeoff situation is required.

Further, the test classes in each browser specific round are executed in parallel but the number of concurrent threads is often kept under five. The choice between parallel classes over methods has been made because single test methods inside a class are not entirely independent and can collide by working with the same objects inside an M-Files vault. However, the number of threads does not fully utilize the potential computing capabilities of the testing machines. Additionally, part of the problem is that some of the test classes are comparative large. This means that the very longest test classes alone will still keep the overall duration long even if more concurrency is added.

A significant duration of time is also spent on analyzing the results of a full test round. With the current process, each failed test case is repeated by manually executing its steps. If the manual test passes then the test case is executed again as a separate automated test. Therefore, the number of failures in a test round can considerably affect the amount of manual work required.

Many test automation related tasks are done manually

A full regression test round is a large manual effort. Some of the manual tasks make very much sense, such as failure analysis and reporting the found defects. However, the test automation process has room for streamlining the tasks that might require only minimal human interaction if at all. Potential problems with manual tasks are that they leave more room for error, they require additional work effort that could be spent somewhere else, and they may be slower to perform.

The very nature of current test automation process is comparative manual because no automation is utilized in starting the tests. That is, a full round of automated tests is started only when a decision to do so is made. By itself this procedure is not a problem but it may be seen as one when combined with the current test automation practices. As per CI principles, continuous regression testing requires the tests to be run very often, after modifications to the application code have been made. Good regression testing would be possible either if the automated tests could be started by issuing a single command when a new build is available or if the tests would be started automatically when changes to version control are detected. With the current process, neither of these options are available so the automated tests are not utilized to their full potential in regression testing.

Moreover, several manual tasks are performed in order to set up the environment for the needs of a test automation round. One such task is the installing of M-Files application and then setting up M-Files Web. This task contains several simple steps, such as copying and running the M-Files installer, and setting the license in the M-Files Admin tool. Also, M-Files Web is launched in the IIS web server by using a dialog in M-Files Admin tool.

The environment setup steps themselves are not very error-prone apart from perhaps copying a wrong installer. Nevertheless, performing the installation operations manually increases the risk of forgetting a required step. Additionally, manual steps increase the duration of the set up operations because desired options have to be selected by using the installer wizard and M-Files Admin UI instead of specifying them as parameters in automated scripts.

Further, test automation code and the test data resource files have to be manually updated from SVN version control system. This step ensures that the latest versions of the automated tests and other required files are used. Yet again, there is a risk of forgetting

to run this step and thus, for example, the wrapper components might not be compatible with the newest version of M-Files Web UI. Also, the test cases might have updates in the SVN repository if they have been modified to respond to changes in the application's features.

A large amount of manual work is invested in analyzing the failures in the testing round and repeating the failed test cases manually and as separate automated tests. However, running the failed test cases again brings value only if the reason for the failure is not already known. Thus, there is a risk that excess manual work is done if the same failing test cases are manually repeated in each testing round. Additionally, the automated reruns of failed tests are started individually by selecting the correct test manually in Eclipse.

Lack of systematic monitoring of testing duration

Data on the product's performance is something that M-Files organization is interested in. Such data could potentially be retrieved by monitoring the duration of the UI tests. Naturally, the duration of a single test case cannot be directly compared with the performance that a user might experience in a similar user case because of the difference of how a human and the wrapper components interact with the UI. However, possible changes in the test durations could indicate changes in the product's or the testing tool's performance.

The problem with the current testing process is rather that the duration data is not utilized than that the data wouldn't be available. TestNG framework records the duration of each test method and each test, and this data is available on the report HTML page that the framework produces. However, the HMTL presentation by itself is not enough for test duration trend analysis because it only contains data on a single test round. Therefore, an effective comparison of test durations between consecutive testing rounds would require a solution to retrieving the duration data from the HTML. Naturally, representations of the durations' trend are simple to produce if such data is regularly collected.

Lack of visibility to the test results

The test results, that is, the reported defects, the full test report, and the TestNG HTML result pages, each receive different level of visibility. First, the best visibility is to the reported defects that are stored to the issue tracking system. There the issues can easily be found based on the metadata associated with them and new defects are monitored and prioritized by responsible team leaders. Additionally, notification emails are automatically sent by the system to inform everyone daily of newly discovered defects.

However, there is a risk that the developer has already moved on to another task and removing the defect is postponed if the results are not available frequently enough. Al-

so, regression defects should be preferably removed as soon as they are found and thus the procedure of cycling the defects through a prioritization process might cause unnecessary delays. A regression defect may be recognized as soon as a related test case is seen to fail and there is a good probability that the developer instantly knows that it was caused because of his or her recent modifications.

Second, a report is made from a full test round, it is stored to the test management vault, and a link to it is distributed to all quality assurance personnel. Creating this report requires detailed analysis of the test results and, in essence, it is a report of what has been achieved by the full UI test automation round. All information in the report is useful to some stakeholders but this type of very detailed analysis may be too rigorous for each testing round.

Also, it should be considered who could make use of this detailed information. Naturally, not all information is relevant to everyone and thus the report may have a too wide range of different details. Currently the report contains useful information but its benefits are not fully utilized. The better utilization would require making the information more visible and putting more thought into selecting its audience. It should be noted that the report is available to everyone in its storage location but the group of people who receive the initial notification of it is quite small.

The third type of test results are the TestNG HTML pages. For each full testing round, multiple result pages are stored in a zip file to the test management vault and it is fairly rare that someone returns to view them afterwards. However, these pages are the main data source for failure analysis but, on the other hand, they lose their relevance relatively quickly when the analysis is complete. Therefore, the developers could maybe benefit from this type or results because it shows what has occurred at the time of test failure. A detailed defect report is naturally as effective to the developer but almost the same information is earlier available in the result HTML page. The data may otherwise also be useful for statistical analysis of failure rates and test durations.

Additionally, some minor points of improvement exist in the different formats of test results. For instance, the test cases are currently named with identifiers that do not describe what the tests do although better descriptions are also available. Further, some of the failure and exception messages produced by the test cases and wrapper components are ambiguous which may make the failure analysis difficult, especially for those who are not very familiar with the test code.

Test quality issues

The failure rates of the automated UI tests are relatively high, on average about 15%. Many of these failing test cases pass successfully when they are repeated separately or at least when executed manually. In fact, only about 5% of the tests really fail because of a valid defect in the AUT. These false positives increase the work required to com-

plete a failure analysis and they may obscure real defects in the application. Also, constantly high failure rates may diminish the motivation to trust and pay attention to the results, and also may make it more difficult to quickly recognize those failures that are caused by defects.

Thus, the failures can be roughly divided to two categories: those caused by defects and those caused by issues in the test cases or in the testing tool. Both of these categories may contain systematic failures and irregular failures. Causes for systematic failures are often easier to pinpoint because they can be reliably reproduced. On the other hand, irregular failures do not happen every time but often have certain conditions that have to be fulfilled in order to reveal the issue. Therefore, there usually is a way to reproduce irregular failures systematically even though they cannot be systematically observed in the testing rounds. Nevertheless, the test cases or wrapper components that cause false positives are the most problematic ones and thus should have high priority for maintenance.

5. IMPROVING WEB USER INTERFACE TEST AUTOMATION

Two most major issues with UI test automation of M-Files Web are the long duration of the full regression round and the large amount of manual work involved in the test automation related tasks. These two issues reduce the effectiveness of test automation because the developers do not receive the feedback fast enough and the manual tasks consume additional testing resources. Additionally, the test quality issues reduce the trustworthiness of the tests and cause even more manual labor. Finally, all produced test results should serve a purpose and relevant information should reach the appropriate stakeholders on time. Data on test durations data is an example of information that the organization regards valuable.

The improvement suggestions presented next in this chapter are first and foremost created to solve or mitigate the presented issues in the web user interface test automation. All the issues are more or less connected with each other and thus solving one issue may also improve the situation with another. The context of the suggestions is the M-Files organization, its product development process, and the tools used in the process. The main focus is on the creation of realistic, achievable solutions that improve the regression testing capability of the automated UI tests but also suit and benefit M-Files organization. Therefore, the solutions do not attempt to fully replicate any specific methodologies presented earlier in this thesis. Nevertheless, the suggestions are inspired by principles of CI, continuous testing, agile software development, and good UI test automation practices.

5.1 Improvement suggestions and their prioritization

The improvement suggestions are prioritized in order to evaluate which suggestions are realistic and beneficial to implement in the scope of this thesis or otherwise in very near future. Two values affect the priority: issue score and effort score. These values are multiplied and the result is the suggestion's priority. The higher the priority the more efficiently the suggestion creates more value.

First, the issue score represents on a scale from 1 to 7 how many issues the suggestion likely solves or mitigates. This score is calculated by adding together the individual priority values of each issue that the suggestion solves or helps to mitigate. These individual issue priority values are presented in Table 5.1 along with the issue identifiers. These identifiers are used later in this chapter when improvement suggestions are mapped

to issues. The issues of long testing duration and high number of manual tasks both have an individual priority value of 2 while the rest of the issues have an individual priority value of 1. Thus, higher priority value means that the issue has higher priority. This differentiation is made because solving the first two issues is regarded most important by M-Files quality assurance management.

Table 5.1 *Issues and their priority values. High priority value means that the issue has high priority.*

Issue	ID	Issue priority
A full round of tests has too long duration in calendar time	Duration	2
Many test automation related tasks are done manually	Manual	2
Lack of systematic monitoring of testing duration	Monitoring	1
Lack of visibility to test results	Visibility	1
Test quality issues	Quality	1

On the other hand, the effort score represents on a scale from 1 to 3 how much effort it requires to implement the suggestion. On this scale, 1 means high required effort and 3 means low required effort. The effort score, in this case, takes into consideration the actual required working effort but also the scope of the solution's ramifications. Therefore, a suggestion that requires high effort from an individual to implement but is easy to adapt to may receive a high effort score. On the other hand, a suggestion that is easy to implement but requires changes in the way of everyday working may receive a low effort score.

High level descriptions of suggestions to improve UI test automation of M-Files Web are presented in Table 5.2. Each suggestion has an issue score and an effort score. Also, each suggestion has a list of identifiers of the issues that the suggestion either solves or mitigates. Finally, each suggestion has the suggestion priority score that is the product of the issue score and the effort score. Thus, high priority score in this case means that the issue has high priority. It should be noted that a low priority score on a suggestion does not mean that it is not worth implementing. Rather, it means that the implementation of the suggestion does not fit into the scope of this thesis. Most of the suggestions affect multiple issues and thus often gain an issue score of 3 or higher. Therefore, the effort score affects the calculation even more. In the context of this thesis, a low priority score may also mean that the suggestion requires more planning and discussion in the organization to really utilize its benefits.

Table 5.2 *Improvement suggestions and their priority scores. High priority score means that the improvement suggestion has high priority.*

Improvement suggestion	Issues	Issue score	Effort score	Priority score
Adding tests as part of automated CI build	Duration Manual Monitoring Visibility	6	2	**12**
Splitting test classes to support parallelism and clarity	Duration Visibility	3	3	**9**
Better failure messages in tests	Manual Visibility Quality	4	2	**8**
Browser rotation	Duration	2	3	**6**
Tracking issues in UI test automation	Manual Quality	3	2	**6**
Test quality control process improvements	Duration Manual Quality	5	1	**5**
Staged build	Duration Visibility	3	1	**3**
Getting the developers more involved in the regression test result analysis	Manual Visibility	3	1	**3**
Use of REST API in tests	Duration Quality	3	1	**3**
Better naming of test cases	Visibility	1	2	**2**
User story impact analysis on automated tests	Manual	2	1	**2**

Next, the suggestions from Table 5.2 are described on a more detailed level and reasoning for listed issues and effort scores are provided.

Adding tests as part of automated CI build

M-Files organization is already using Teamcity continuous integration server for automated integration testing of M-Files API so continuing that work with UI tests is relatively straightforward. It is clear that a large number of manual tasks in testing environment setup can be automated and added as steps to a build in a CI server. Most of these tasks are also faster when they are automated. Additionally, a fully automated build makes the test round duration shorter by reducing downtime between different steps. Frequent CI builds also increase the visibility to the test results by utilizing dashboards, notifications, and metrics provided by Teamcity. Also, Teamcity tracks the duration of builds and similar data is also available on test case level.

The effort score for this suggestion is 2 because the use of Teamcity is already established in the organization. Therefore, several re-usable steps already exist for setting up

the testing environment. On the other hand, the implementation requires setting up a testing infrastructure including hardware and software, and ensuring the compatibility between the UI testing tool and Teamcity. However, the implementation mainly requires only working effort rather than adopting new working methods. Effectively, only the manner of executing the tests is changed while the result reports generated by TestNG framework and the issue reports remain the same.

Splitting test classes to support parallelism and clarity

With the current implementation, only one thread at a time can safely work on a single test class. Thus, splitting a test class into, for example, two smaller classes allows two threads to access those tests simultaneously. Currently, TestNG classes that contain a large number of UI test methods can take several hours to execute by a single thread. Additionally, it is difficult to gain an understanding of what type of test cases these large test classes contain. Therefore, splitting such test classes into smaller logical components supports both better parallelism and clarity.

It should be noted that with this method the duration of a test round can be reduced only if the testing computers in the platform have enough computing power to support multiple concurrent browsers. Also, running test methods instead of test classes in parallel would in theory achieve similar or better results in duration reduction without splitting the test classes. However, the test cases are not entirely independent so there is a risk of two tests trying to access the same object simultaneously or in a wrong order.

The effort score for this suggestion is 3 because splitting large test classes requires merely identifying logical subsets inside them. Then, naturally, the methods have to be moved inside newly created classes. The tests itself do not change and thus the splitting does not affect any testing processes. However, the Excel files containing the test data must also be split which causes some extra manual work. Nevertheless, this suggestion provides continuous benefits by investing resources on a one-time effort.

Better failure messages in test cases

The test result failure analysis could be made easier by producing error messages that are as exact as possible. The clarity of the error messages is often adequate if the test fails at the verification in the end. On the other hand, sometimes the tests fail because an exception is received from the wrappers. The exception messages from the wrappers range from custom messages to detailed exceptions from Selenium. The Selenium exceptions often reveal the root cause, for instance, a certain HTML element was not found on the page. However, the person doing the failure analysis may not be able to recognize the element by the selector that is mentioned in the error. Thus, adding custom messages to exceptions may serve the clarity better if they are precise enough. For example, the error message "item X was not found in the list" should rather be "item X was not found in the list of possible values for the property Y in the metadata card".

Naturally, also all custom error messages should be made more clearer, not only in those cases when the error originates from Selenium.

Therefore, improving the failure messages could reduce the manual work in failure analysis because the error may be identified without looking at the test code. On the other hand, this also increases the visibility to the test results because a wider range of people can understand the failure messages more easily. The improvement also applies to test quality because possible defects may be identified faster from clearer results and the risk of accidently overlooking a defect decreases.

The effort score for this suggestion is 2 because it requires some effort to locate the areas in the test tool code where and how error messages should be improved. However, otherwise the suggestion does not affect the testing and development process. This suggestion may be implemented gradually, for instance, by reviewing the error messages in those areas where the code is under maintenance for other reasons. Other way is to specifically target those areas that seem to have unclear messages as they are gradually revealed by normal testing rounds.

Browser rotation for UI tests

The full regression testing round has a very good browser coverage but this greatly increases the testing duration. However, rotating the browsers between consecutive builds could gradually achieve the same coverage. This suggestion is dependent on the improvement that the testing rounds are run more often, for instance, by adding the tests as part of a CI test build that is run for each new M-Files build. Therefore, achieving full browser coverage, for example, across five frequent consecutive CI builds is better than achieving full browser coverage for a single build that is not run frequently. The browser rotation may greatly reduce the execution duration of a single testing round and should not affect the coverage in the long run when compared with the original testing rounds.

The logic for the rotation is visible in Table 5.3 where the full regression test suite is divided into five test sets that each use different browsers. It should be noted that this suggestion favors the fast detection of defects that are not dependent on the browser over browser specific errors. However, frequent builds should uncover the browser specific errors eventually but not necessarily as soon as possible. For instance, in worst case scenario, a Chrome specific defect in test set 4 will be not be detected until in build number 5 even if it was introduced already in build number 1.

Table 5.3 *Browser rotation that achieves full browser coverage for five different browsers across five consecutive builds. The browser used for each test set is rotated between consecutive test builds.*

Test build number	Browser for test set 1	Browser for test set 2	Browser for test set 3	Browser for test set 4	Browser for test set 5
#1	IE9	IE10	IE11	Firefox	Chrome
#2	Chrome	IE9	IE10	IE11	Firefox
#3	Firefox	Chrome	IE9	IE10	IE11
#4	IE11	Firefox	Chrome	IE9	IE10
#5	IE10	IE11	Firefox	Chrome	IE9

The effort score for this suggestion is 3 because the browsers used in tests are simply defined in the TestNG XML file on test level elements. Thus, the browser rotation can be achieved by defining TestNG XML files that only differ in their browser configuration. Then, these different XML files are rotated between consecutive testing rounds.

Tracking issues in UI test automation

Issues in the UI tests are sometimes caused by known errors in the test automation tool. These issues can be in the page classes, wrappers, or in the test cases themselves. It would be beneficial to keep track of these known issues so that the same issue is not analyzed multiple times and the issue can be prioritized for maintenance work. Therefore, it is especially important that these issues are made visible.

Similarly, reported defects in the AUT could be mapped to the affected failing test cases. This would make the failure analysis easier because searching by the failed test case identifier would bring forth the issue report. This could also aid in automatic verification that the issue has been correctly removed by executing the listed test cases.

Good tracking of issues in UI test automation could also reduce manual effort in failure analysis because the known issues would not have to be re-analyzed and re-run. Additionally, better visibility to the issues would aid in having a clear workflow for correcting them. Therefore, this suggestion could improve the maintenance work and thus test quality.

The effort score for this suggestion is 2 because the issues are already kept track of but the process should simply be made more visible and organized. Nevertheless, some additional effort is still required for converting the current system into a proper, visible workflow. For instance, the platform of the tracking system could be the Test management vault but that would require planning and prototyping the metadata structure. A simple solution would be to keep track of issues in an Excel file and have that commonly available.

Test quality control process improvements

The high number of false positives in UI tests indicates that the quality process for the automated tests in not in control. That high number also greatly diminishes the ability to detect defects in test results of a regression round by performing a quick analysis. Therefore, the currently failing UI tests require a systematic analysis of the causes for the failures.

This systematic analysis should affect all current TestNG test classes in UI test automation. Individual test classes contain similar test methods and thus their failures are often related to each other. Additionally, the analysis process is kept more clear when whole test classes, instead of individual test methods, are put under maintenance. It should be noted that this systematic failure analysis is performed in order to take the test automation process under control, that is, to ensure that the UI tests are robust enough for the continuous execution in CI. Therefore, the test classes are kept under maintenance until the class as a whole can be deemed ready for CI testing builds.

First, the failure rates of all test classes should be measured. Then, the failure analysis should be performed on the test classes that either contain a small percentage of failures or a small number of failures. All valid defects in the AUT should be reported and then maintenance operations should be performed until no false positives remain. After the maintenance, these test classes will form a known set of tests that have very low failure rates. Therefore, these test classes can be more effectively used in CI.

The quality control process should then be extended to the test classes with high failure rates. The quality control process can be seen in Figure 5.1. At this point, also the failure rates of the repaired test classes in CI should be actively monitored to ensure that the test automation process remains in control. In addition to correcting the false positives, also the feature coverage and quality of the test methods should be assessed. Accordingly, new tests should be created and unnecessary existing tests removed. Also, it could be beneficial to monitor what types of regression defects get past test automation and try to augment the test suites based on that information.

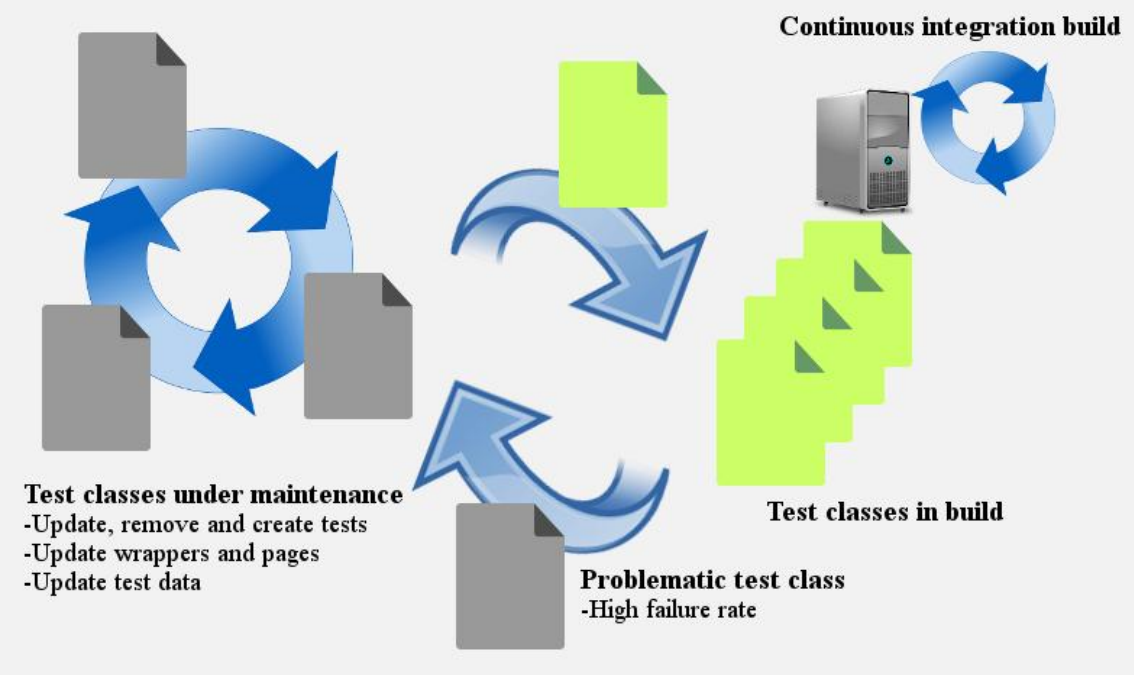

Figure 5.1 *Quality control process where test classes with high failure rates are analyzed. Maintenance is performed on these classes until they are robust enough to be added to the continuous integration build.*

The effort score for this suggestion is 1 because it requires great effort to identify and correct the causes for the failures. The process requires resources and a systematic approach. Additionally, the work requires good knowledge on the tests under inspection and very likely also on the element structure of the related UI components.

Staged build

A staged build would first run a shorter set of smoke tests and then start a longer duration regression testing build. This would quickly provide some feedback to the developers and aid in correcting any detected issues fast. Therefore, a staged build would effectively reduce the duration for receiving the first test results for a build. Additionally, this staged build could also increase the visibility to the test results because a short list of commonly known tests could be easier to follow by the developers and other stakeholders.

However, M-Files is such a mature application that modifications to the code rarely cause so critical failures that they would be detected by very basic smoke tests. Thus, it is not certain if automated smoke tests would add much value to the testing build. On the other hand, the criticality of such failures might still justify the utilization of a smoke test set in a staged build.

The staged build can also be implemented as a set of multiple parallel builds. With enough computing capacity, and an adequate number of computers, a regression test

round could be split into smaller parallel rounds. With such a setting, for instance, regression tests could be executed for different browsers in parallel, each browser as an independent build.

However, the effort score for this suggestion is 1 because suitable test cases for a good smoke test set are to be identified, and possibly, to be developed. Additionally, the implementation of a staged build with a limited number of testing computers would require more study because, for instance, the builds should always execute in a correct order. On the other hand, increasing the number of available computers or virtual machines for testing is also not reasonable until the benefits of a staged build are more clearly understood.

Getting the developers more involved in regression test result analysis

Effectively, the developers know best the modifications they have made to the code and thus they may also have the best understanding of the ramifications of those modifications. Therefore, it could reduce the manual effort required in failure analysis if the developer identifies errors caused by his or her recent modifications. This practice would naturally also increase the visibility to the regression test results.

The effort score for this suggestion is 1 because the implementation would require changes in the current way of working. For instance, reading automation test results does not belong into current responsibilities of the developers. Additionally, most of the developers are not familiar with the used test automation tool or the automated test cases. Therefore, analyzing the test results would require first learning the TestNG result format and have an understanding of what reasons the tests can fail for. However, developer involvement could diminish the clarity of responsibilities in result analysis, that is, issues that no developer recognizes as their own might be overlooked. Thus, the ultimate responsibility for analysis might be better to remain with the testing team, even with developer involvement. Nevertheless, developer involvement with proper training and mindset could bring benefits to faster removing of defects.

Use of REST API in tests

Some of the test classes and individual test cases require configuration initialization by using the configuration page UI in the application. These initialization steps must succeed in order for the tests to run correctly and thus the steps should be reliable. Running these steps by using the UI may sometimes fail because of instability in Selenium startup and, in worst case, this can cause multiple test cases to be skipped. However, the configuration UI has its own test cases and using it in the initialization is only the precondition of the test. Thus, these configuration steps could be executed by using M-Files REST API to make them more reliable and thus reduce unnecessary failures. Additionally, utilizing the REST API could reduce the testing duration of some test classes because the operations are considerably faster when compared with performing them by

using the UI. This is especially the case in test classes where each test case has to perform setup operations.

The effort score for this suggestion is 1 because communication with the REST API requires implementation of a class that utilizes HTTP requests and can handle the inbound and outbound JSON formats. The implementation requires also study of the REST API. Additionally, this suggestion requires modifications to the initialization steps of affected test cases and test classes. However, existing third party libraries can naturally be utilized in basic HTTP and JSON operations.

Better naming of test cases

The UI test cases are currently named with identifiers that do not describe what the tests do. On the other hand, the tests have descriptions in their TestNG annotations. However, the name of the test is used in the test results and thus a descriptive name would greatly improve the readability of these results. For instance, the names could be derived from the existing descriptions in the annotations. However, some of the descriptions may have to be made shorter to make the names concise. On the other hand, the identifiers are useful for quickly referencing to a test method in daily communication Therefore, keeping the identifiers as part of the test method names could be beneficial.

The effort score for this suggestion is 2 because it is a relatively large effort to modify the names of all test methods. The names would also have to be changed in the test data Excel files that use the test method names as identifiers for the test cases.

User story impact analysis on existing automated tests

The user stories of a sprint are decided in the sprint planning meeting. Thus, there is a reasonable opportunity to analyze the impact of these user stories to the existing automated test cases. Especially changes to existing functionality or to the UI can affect the tests and wrapper components. Therefore, this analysis could make the test maintenance a more predictable and controlled process and also aid in the failure analysis. In best case scenario the affected tests would be put under maintenance and cycled out of the regression round until they are synchronized with the changed functionality. Naturally, the user stories that consider M-Files Web have the most significant impact on the automated UI tests.

The effort score for this suggestion is 1 because it is not clear how difficult such analysis may be. The analysis may require good knowledge of both the UI element structure of the AUT as well as the automated tests. Otherwise there is a risk that some test cases or wrapper components are put under maintenance in vain. However, the mere knowledge of upcoming changes can be enough to improve the failure analysis and test maintenance process. That requires either good communication from the development teams or active participation to the sprint planning meetings by the testing team.

5.2 Implemented improvement suggestions

The following improvement suggestions were implemented in the scope of this thesis:

- Adding tests as part of automated CI build
- Splitting test classes to support parallelism and clarity
- Browser rotation for UI tests

Additionally, the following improvement suggestions have been partially implemented:

- Test quality control process improvements
- Use of REST API in the tests

The first three implemented suggestions all have high priorities and effectively they all are required to run the tests smoothly as part of CI. The first suggestion acts as an enabler for running the tests as part of CI. On the other hand, the next two are ensuring that the testing build duration is kept short enough so that the feedback is provided in an acceptable time.

The first partially implemented suggestion considering test quality has a relatively high priority but it requires a great effort. Nevertheless, in order to increase the value of the three implemented suggestions, its implementation has been started and the first phase has been finished. The other partially implemented suggestion, related to REST API, was at first prioritized higher and its implementation was started by creating a class for communicating with M-Files REST API. However, the implementation was left pending because other improvement suggestions were considered having higher priority.

Next, these implemented improvement suggestions are described in detail.

5.2.1 Adding automated tests as part of CI

The implementation of this suggestion requires setting up an infrastructure that hosts computers with suitable hardware capabilities and required software. Additionally, the testing tool requires some modifications to enable running the tests without any manual tasks. Finally, a build has to be configured in Teamcity CI server.

Setting up the hardware and software infrastructure

The existing computers currently used in UI test automation are a combination of physical computers and virtual machines. However, these computers cannot be utilized in the new setup because they reside in a different network. Additionally, the current UI test automation practices are continued and will exist alongside the new CI practice during the setup phase. Therefore, a whole new infrastructure for UI test automation has to be created.

The new infrastructure is implemented in M-Files network on top of an existing vSphere ESXi (Vmware.com 2016) installation that allows the partitioning of a physical server into multiple virtual machines. The new setup contains five virtual machines that are created as copies from existing virtual machine images and then migrated to the vSphere platform. These virtual machines will form the Selenium Grid, one acting as a hub and the rest acting as nodes. The details of each virtual machine can be seen in Table 5.4.

Table 5.4 *Details of the virtual machines in the implemented infrastructure that is used in web user interface test automation builds.*

Computer	**Operating system**	**Browsers**	**Software**	**Hardware**
Teamcity agent	Windows 8.1	Firefox Chrome IE11	Teamcity build agent IIS M-Files server Java Maven Selenium Chrome driver IE driver	8GB RAM 4 processor cores
Selenium node#1	Windows 7	Firefox Chrome IE9	Java Selenium Chrome driver IE driver	4GB RAM 4 processor cores
Selenium node#2	Windows 7	Firefox Chrome IE10	Java Selenium Chrome driver IE driver	4GB RAM 4 processor cores
Selenium node#3	Windows 8	Firefox Chrome IE10	Java Selenium Chrome driver IE driver	4GB RAM 4 processor cores
Selenium node#4	Windows 10	Firefox Chrome IE11	Java Selenium Chrome driver IE driver	4GB RAM 4 processor cores

First, one of the machines, running Windows 8.1 operating system, acts as a Teamcity agent that is registered to an existing Teamcity server on M-Files network. This enables the communication between the agent and the server. This communication is required for starting and controlling the CI builds. This machine also acts as a Selenium hub and thus requires Java installation and Selenium executable. Then, the machine will host M-Files server and M-Files Web, and therefore an IIS installation with a set of required features is needed. Finally, the virtual machine also requires a Maven installation in order to compile and run the tests.

The other virtual machines, two Windows 7 machines, a Windows 8 machine, and a Windows 10 machine act as Selenium nodes. Thus, they also require Java installation and Selenium executables. Additionally, these Selenium node machines require driver executables for Chrome and Internet Explorer while the Firefox driver is a built in feature of Selenium. All virtual machines have Chrome and Firefox browsers, and additionally each has some version of IE. The first Windows 7 machine has IE9, IE10 is installed on the other Windows 7 machine and on the Windows 8 machine, and finally, the Windows 10 machine has IE11. Additionally, the Selenium hub machine has IE11 and can also act as a Selenium node if required.

Each of the Selenium nodes are configured to allow five concurrent instances of a specific browser. However, the number of maximum concurrent sessions, that is, the number of parallel browser processes, is also limited to five. Effectively, this means that each Selenium node virtual machine can have a maximum number of five concurrent browser processes independent of the type of the browser. These numbers can be adjusted by providing different command line parameters to Selenium. However, the number of concurrent browsers should be kept at a number that does not consume too much memory and computing power. The Selenium Grid status can be viewed in a Selenium hub dashboard web page. That dashboard is visible in Figure 5.2.

Figure 5.2 *Selenium Grid dashboard web page that shows the available browsers on the nodes that are registered to the Selenium hub.*

Enhancing test environment setup initialization in the testing tool

The initialization methods in the testing tool also require some modifications in order to fully remove all manual steps in the environment setup. Originally, restore operation for

a test vault was done by calling a console application in initialization methods of the TestNG test classes. However, this console application required the M-Files server to contain a test vault. The console application made a backup of that vault and that was used as a basis for new test vaults. This practice required the use of M-Files evaluation installation that automatically installs a sample vault that can be used in testing.

A new approach is to use a PowerShell script instead of a console application in the initialization method. This new script allows its caller to define the name of the backup file as a parameter and that backup file is used to restore a test vault. Therefore, the script enables the use of different test vaults if their backup files are provided. This also removes the requirement for an existing vault on the server. Additionally, the script is not required to be compiled and thus it is more flexible as a resource file within the test automation tool.

Similarly, also the creation of test users was originally possible by using a console application. However, the console application was not often called because the sample vault integrated in the evaluation installation already contained some test users. Some special cases were not also handled by this application, for instance, if the users were created by restoring a vault from a backup file. In that case the users could not be created again by the application but they were left without a required user license.

Thus, also this application was replaced by a PowerShell script. The script reads user account information from the test data Excel files and creates the users accordingly. The script is also able to modify information of existing users in case of that they are already present in the server. Therefore, this new script allows a more flexible way to create test users independent of any existing users in the server.

Configuring the build in Teamcity CI server

Creating a build configuration in Teamcity is done by defining version control settings, build triggers, and build steps. Additionally, the build can make use of pre-defined and user-defined build parameters (Jetbrains.com 2016b). These features of Teamcity are utilized in order to create a build for running automated UI tests for M-Files Web.

The version control settings are defined for a project in Teamcity. Then, individual builds configurations can be created under this project. Therefore, a new project for UI test automation is created in Teamcity and a connection to SVN version control system is defined. The M-Files SVN repository has source code and all required resources for both M-Files and UI test automation. Only required parts of the version control repository structure can be fetched to the Teamcity agent by using version control system checkout rules. Thus, three different folders are fetched from the repository: all code and resource files for UI test automation, utility scripts, and M-Files trunk source code. Source files for M-Files trunk are fetched so that the modifications in the application code can be tracked and then mapped to possible failing tests. The source code for M-

Files is not required for compiling because this new Teamcity build configuration is created only for UI testing.

The build configuration requires a trigger in order to automatically start the build when certain conditions are fulfilled in the SVN repository. Thus, a new trigger is created so that a build starts when a certain comment is entered to the version control system along with a commit. This comment is used in an automatic commit to SVN every time the installer for a new M-Files build is available. The automatic commit always modifies certain definition files in M-Files trunk code. The build configuration can only monitor changes to files that are included by the checkout rules and that is another reason why M-Files trunk source code is fetched to the build.

With the defined version control settings and the trigger, a new Teamcity build will start every time a new M-Files trunk build is available, and the required files are checked out to the Teamcity agent virtual machine. Then the build starts executing the builds steps. All build steps, apart from running the tests, are executed as PowerShell scripts. Most steps perform some setup operations using the M-Files API and thus the using PowerShell scripts is a valid choice. Teamcity has a PowerShell runner and the scripts are flexible for small tasks that may have to be modified from time to time. These build steps are as follows:

1. Download M-Files installer: This step executes a PowerShell script that copies the installer for the new M-Files build from the builder computer in M-Files network. The used script is defined by a build parameter. By default, the installer for the first M-Files build which contains the commit that triggered the Teamcity build is fetched. By defining a different value for the build parameter, this script can be substituted with another script that copies the installer for the most recent M-Files build. This is useful, for instance, when the build is started manually and there are new commits created after the most recent M-Files build. Otherwise in this case, the build would fail because there are no M-Files builds available that contain the most recent commit. This step was re-used from the existing build configuration for M-Files API NUnit tests.

2. Choose/rotate TestNG XML file: This step executes a PowerShell script that chooses the used TestNG XML file for the build. The used file rotates, based on a certain logic, so that two consecutive builds always use different browsers in the UI tests. The logic for this step is explained in chapter 5.2.3.

3. Install M-Files: This step executes a PowerShell script that installs M-Files by using the downloaded installer. The script command installs a fresh M-Files server, M-Files Admin, and M-Files Desktop. The step was re-used from the existing build configuration for M-Files API NUnit tests.

4. Add login account for current user to M-Files server: This step executes a PowerShell script that adds a login account for the current Windows user to the M-Files

server. A login account is required for certain M-Files API client operations. This step, at the moment, is not a requirement for the UI tests. The step was re-used from the existing build configuration for M-Files API NUnit tests.

5. Set M-Files Server license: This step executes a PowerShell script that installs an M-Files license to the server. The license code is received as a build parameter. The license is needed so that M-Files application can be used after the trial period of 30 days. The step was re-used from the existing build configuration for M-Files API NUnit tests.

6. Configure M-Files Web to Default www-site: This step executes a PowerShell script that configures M-Files Web application to IIS default web site. The step was re-used from the existing build configuration for M-Files API NUnit tests.

7. Restore test vaults: This step executes a PowerShell script that one by one restores those test vaults that are specifically marked for restore operation in the used TestNG XML file. The first 20 vaults in the XML file are marked to be restored because otherwise the initialize methods of the corresponding test classes would launch concurrently as soon as those 20 first parallel test classes begin. This step is required because the vault restore operations take a very long duration if several of them are launched at the same time in the server. However, it is unlikely that several test classes will finish execution at the same time and thus only the test vaults of the first 20 concurrent test classes are restored by this script. The rest of the test vaults are restored by normal initialize methods of their test classes when their execution begins at some point during the build. The number of test vaults to be restored by this script can naturally be altered by modifying the TestNG XML file.

8. Update product version parameter to TestNG XML file: This step executes a PowerShell script that modifies the used TestNG XML file's product version parameter with the current M-Files build number. This information is needed in initialize methods of some test classes in order to set certain configurations in the Windows registry.

9. Run Web UI tests: The tests are executed as a Maven step by using Surefire plugin (Apache.org 2016b). The Surefire's test goal is bound to the test phase of the Maven default lifecycle. The path to the TestNG XML file has to be provided to Surefire by using Maven's pom.xml file. The path can be given as a command line parameter when the TestNG XML file path is defined as dynamic parameter rather than a fixed path in the pom.xml file. Therefore, this step consists of a single command to Maven to first execute the clean phase including all previous phases of the clean lifecycle and then to execute the test phase including all previous phases of the default lifecycle. Finally, the path to the TestNG XML file is read from a build parameter and given as a parameter to the test phase.

10. Create result folder: This step executes a PowerShell script that creates a folder for storing the TestNG result HTML, screenshots, and other related files. The folder is named based on the running number of the Teamcity build.

11. Copy test results: This step executes a PowerShell script that copies the TestNG result files to the folder created in previous step.

12. Uninstall M-Files: This step executes a PowerShell script that uninstalls all M-Files applications of the related build from the computer.

13. Delete M-Files installer: This step executes a PowerShell script that deletes the M-Files installer from the computer so that it does not remain in the disk and consume space.

This build configuration effectively automates the UI testing build and the manual work begins with result analysis only after the build has finished. Additionally, the dashboards and notifications provided by Teamcity are utilized. By viewing the build configuration dashboard, anyone can easily see the results of the recent builds. For instance, the number of failed test cases and the build duration can be seen at a glance. The build result format can be seen in Figure 5.3. More details are available in the form of a complete build log. The notifications of failed builds are also automatically sent by email to all developers who have made modifications to the code.

Figure 5.3 *Teamcity dashboard shows a quick overview on the test results, changes to code, and build duration. This figure displays three builds.*

5.2.2 Splitting test classes to support parallelism and clarity

This suggestion is effectively implemented by identifying logical test method categories in large TestNG classes and then moving the test methods to new classes by their category. In practice, the original names of the large test classes are used as top-level categories and new folders are created for them in the testing tool folder structure. Then, the newly split test classes are simply placed under these new category folders. This can be seen in Figure 5.4. At this point, the splitting is made only for test classes that have very long durations. Naturally, such test classes tend to contain a large number of test methods.

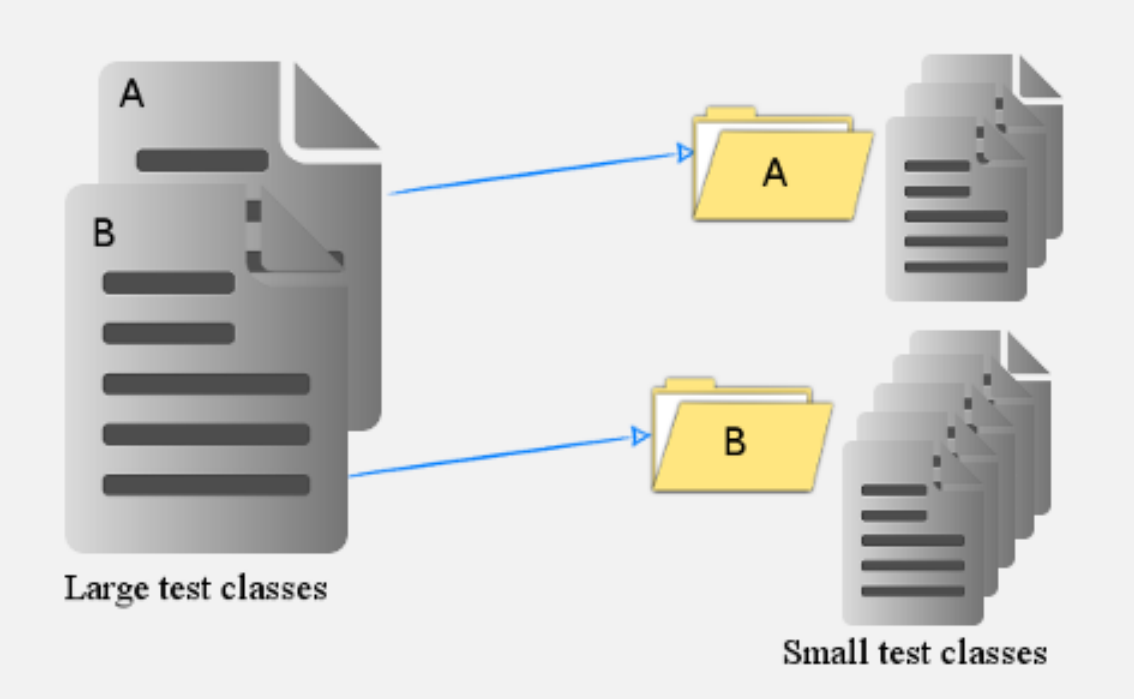

Figure 5.4 *Dividing large test classes into smaller classes. The original test class names will remain as category names of the folders in the new structure.*

The implementation is approached by first running a full round of tests to identify the test classes that have long durations. All test classes are executed by using Chrome browser so that the results are comparable with each other. This is done because often the duration of the same test class with two different browsers may differ significantly. It should be noted that a one-time measurement may be affected by some situational factors in the environment and the results may not represent the average duration of the test classes. However, this is acceptable because the results are merely used as a starting point for identifying the most problematic test classes. Thus, more test classes can be divided into smaller parts if the need for that arises.

At this stage, all test classes that have a duration of over three hours are divided. Therefore, three hours is the minimum duration for the UI testing builds because a single test class is executed by a single thread. Three hours is an acceptable duration because that often enables the test results for a build to be available during the same working day. Thus, the test classes are split so that the test methods form logical categories where the number of test methods in each category ranges between 10 and 40. However, also the new test classes should not exceed the three hour duration.

The categories are formed by manually identifying keywords based on the test methods' descriptions. The keywords can be M-Files features, UI components, or other themes that are common to many test methods. In some cases, a test method would match the theme of multiple test classes. At that point, the test class which seems like the best match is chosen for the test method. However, additional categories for the test methods

can be assigned by using the TestNG grouping annotations if required. It should be noted that the splitting could have been implemented by assigning the test methods into groups instead of dividing them to new classes. That solution would not have addressed the lack of clarity in the oversized test classes. The smaller test classes make it easier to comprehend the contents of each class and thus it may also be easier to judge if there are gaps in the test coverage of some features.

The benefits of splitting the large test classes becomes evident when they are executed in parallel. Smaller test classes utilize the available computing capabilities better because more concurrent threads can be added to execute the tests. However, the execution order of the test classes is also very important if there are more test classes than available threads. The minimum overall duration for the testing build is achieved by executing the test classes in descending order based on their duration. This can be seen illustrated in Figure 5.5.

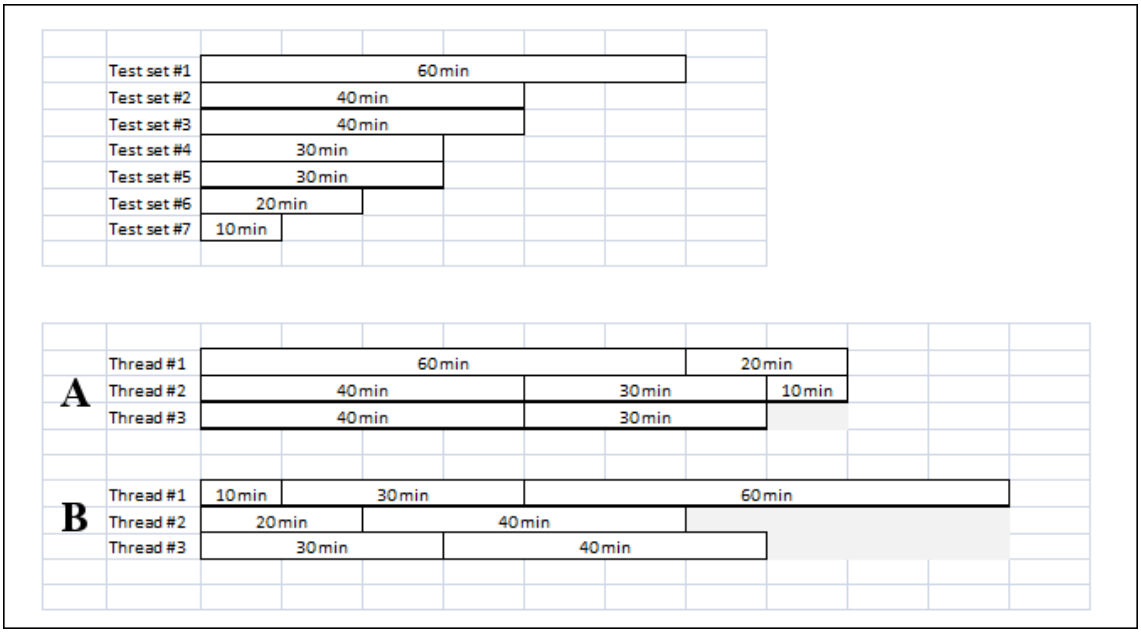

Figure 5.5 *The order of the test sets affects the overall duration when the tests are executed by using multiple threads. Option A is the optimal solution to run the seven test sets by using three threads, that is, the order of the sets is from longest duration to shortest duration. Option B is an example of a less optimal solution.*

Naturally, the durations of the test classes should be monitored and the testing order should be based on that information. The test classes are executed in the same order that they are listed in the TestNG XML file so the order can be altered by modifying the file. It should be noted that in most cases the test classes do not have to be listed in the exactly descending order based on their duration. It is most important that test classes that are considerably longer than the other test classes are started in the initial batch of concurrent threads.

5.2.3 Browser rotation for UI tests

This suggestion is implemented by defining multiple TestNG XML files that contain the same test classes but for different browsers. Each XML file contains some tests for each supported browser. This makes sure that at least login functionality and some subset of features are always tested for all supported browsers, although the features for each browser vary from build to build. The logic for defining the browsers for the TestNG XML files was presented in Table 5.3.

In practice, the browser rotation for consecutive testing builds is implemented by a PowerShell script in the Teamcity build step that was mentioned in chapter 5.2.1. This step requires the use of some custom and pre-defined build parameters. First, a build parameter that contains the file paths to all TestNG XML files used in the rotation is required. The file paths of individual files are separated by a separator character, in this case by an exclamation mark. This parameter makes it easy to define and modify the used set of XML files in the rotation. Then, another custom build parameter is used to store the file path of the selected TestNG XML file for the build. The value for this parameter is defined by outputting the file path as a Teamcity service message in the PowerShell script.

The script makes use of the running build number that is read from a pre-defined build parameter. The script chooses the XML file by dividing the build number by the number of XML files in rotation. The remainder of this division is used to define the used XML file for this build. Therefore, zero means the first XML file, one means the second file, two means the third file, and so on. However, the rotation script cannot determine if the previous build has executed any tests. Thus, builds that fail before running any tests still use up one rotation turn.

Additionally, the script allows to override the used XML file instead of using the rotation. The override is enabled by setting a specified build parameter flag as true. When the override is enabled then the TestNG XML file path is read from another build parameter that is especially defined for the override purpose. The overriding is especially useful if the build is run as a manual build with specific requirements. Such requirements can be, for instance, a specific browser composition or some specific set of tests. In that case, the build parameters can be specified for that individual custom build and those settings do not affect the usual regression test build cycle. However, it should be noted that running such custom builds still uses up a rotation turn similar to the failed testing builds.

5.2.4 Test quality control process improvements

The implementation of this suggestion has been started by performing an initial analysis of the failure rates of the test classes. The analysis was performed on the new divided

test classes formed by the split process described in chapter 5.2.2. A full testing round using Chrome browser and containing all the test classes was executed, and the failure rates as well as skip rates were written down. In most cases, a skipped test method means that an object specified in the test data could not be located in the test vault. Thus, the test cases can sometimes even be skipped because of defects. On the other hand, the test data can be invalid or there is an issue with the test. Therefore, both failed test cases and skipped test cases often must be investigated.

In the initial analysis round, all test classes that have a combined failure rate and skip rate of over 10% are deemed problematic. Such test classes are taken under maintenance and analyzed before they are applied to the CI build in Teamcity. However, an exception to this rule is the situation when the combined number of failing and skipped test cases in the test class is under five. In such cases the test class can be accepted to the build.

Still, the broken test cases, that produce false positives, are excluded from the build until they are functioning correctly. It should be noted, on the other hand, that test cases that fail because of a defect in the application should not be excluded from the CI build merely to make the pass rate higher. Skipping such test cases may provide an overly positive image of the state of the build. However, there should be a fast way to recognize these known issues in the failure analysis. For instance, Teamcity marks newly failed tests separately in the test results along with listing the old failures. Thus, a rapid failure analysis can be performed only on the new failures.

After the test quality analysis, test classes containing about 450 test cases out of 1700 were accepted to the CI build. In the next phase of this improvement suggestion, the failures in the rest of the test classes are investigated. Gradually more test classes will be added to the build when they are deemed trustworthy. At this point, work should be done in order to get the CI build gain the trust of the developers and other stakeholders. The results of the further analysis to come are out of scope of this thesis.

5.2.5 Use of REST API in tests

This initial steps have been taken to implement this improvement suggestion by creating a class for communication between the testing tool and M-Files REST API. At this point this implemented class provides methods for requesting authentication tokens to M-Files server and vaults. Also, a method has been implemented to send an HTTP GET request to a vault which results in a JSON string representation of the requested resource.

The current solution basically enables an easy way to get information from the test vaults and the server. For instance, getting the identifier of the test vault by using the new solution is faster and less error-prone than retrieving it by logging in to the configu-

ration page. However, each case where data from the REST API is utilized has to be implemented separately because the JSON representations of the available resources types naturally differ. Thus, the work with this suggestion may continue by gradually expanding the public interface of the class further. For instance, additional methods may provide access to different M-Files resources in the form of objects and data types that can be understood and utilized by the test classes. Moreover, communication with the REST API should optimally be encapsulated so that the user of the interface does not need to handle the authentication tokens.

5.3 Evaluation of the implemented improvement suggestions

The aim of this thesis was to solve or mitigate the identified issues in the UI test automation process and in the UI test automation tool for M-Files Web. For that reason, several improvement suggestions were made and some of these were implemented. Now it is evaluated how well the issues were solved by the implemented improvement suggestions. Additionally, the limitations of the current implementations are addressed.

The first issue was the very long calendar time duration of a full testing round. The current Teamcity UI test automation build takes about 3-5 hours to finish depending on the used browsers. The failure analysis practices are not yet fully established but currently about 15-60 minutes are spent on failure analysis depending on the number of the new, not already known failures.

The current CI build implementation is very fast when it is compared to the original full testing round that takes several days to finish. However, the two durations are not comparable because the current build has only included about 450 test cases out of the existing 1700. Still, the testing build, by far, does not fully utilize the computing capabilities of the current infrastructure setup. Therefore, it is very likely that the duration of the build will not increase at all even if several new test classes are to be added to the build. This is mainly possible because the new smaller test classes can be better executed concurrently.

The main reason for the short duration of the testing build is the rotation of browsers between consecutive builds. Effectively, the rotation reduces the testing build to approximately one fifth of the size of the original full testing round. Often there is at least one new M-Files build per day so with browser rotation the full browser coverage is achieved gradually across multiple builds in a few days. Practically, the implemented testing builds are comparable to the individual browser specific testing rounds that formed the original full testing round. By these standards, the original testing rounds have now taken the form of automated builds that have tests using multiple different browsers instead of one specific browser.

It should be noted that the current implementation has some issues regarding Internet Explorer browser. Currently, the test automation tool has no support for selecting specific IE version for the tests. This selection has been achieved before in the full testing round by having multiple Selenium hubs which each have only had one registered node with a specific IE version. Such hubs have been used in executing browser specific tests rounds. This is not feasible with the current infrastructure that has multiple different IE version nodes registered to a single Selenium hub. Additionally, some single test cases sometimes take a very long duration on IE version 9 which often also makes the overall duration of the build longer. This is an unfortunate issue because the whole build duration can be affected by a small number of individual test cases.

Also, one high priority issue was that the UI test automation process had several manual tasks. Effectively, the Teamcity build steps have now automated all test environment setup tasks. Also, the build trigger now automatically starts the testing build as soon as a new M-Files build is available. Thus, the testing build can run from start to finish unattended. Naturally, both failure analysis and defect reporting are still manual tasks that must be performed after the build has finished.

On the other hand, lack of systematic monitoring of testing duration is an issue that has not been entirely solved yet. Teamcity gathers duration data of all builds and individual test cases that it runs but, unfortunately, the duration data contains errors if tests are executed concurrently. Drastic changes in performance can naturally be easily detected by monitoring the overall duration of the builds. However, the produced TestNG HTML result pages contain correct durations for each individual test case but utilizing this data would need further work.

Another issue was the lack of visibility to the UI test automation results in the organization. The visibility has been significantly improved because now Teamcity web dashboard provides a quick overview of the UI test automation results. It is also convenient that the results of M-Files API test builds and UI test automation builds can now both be seen in the same dashboard. Additionally, email notifications are sent to the developers that have made commits to the version control if the build has new failures. However, browser rotation in the build causes Teamcity to regard reoccurring browser specific failures always as new failures. This causes developers to receive notifications from failed builds that do not have any new failures.

The HTML result page provided by TestNG framework is currently used only for failure analysis and is stored in the file system of the Teamcity agent virtual machine which has only limited access. This is because the UI test automation failure analysis process of the new CI build format is not yet fully established. For instance, the responsible persons for failure analysis have not yet been nominated. Thus, full utilization of the new UI test automation build still requires clarification of responsibilities and actions so that failures in build are always investigated and tasks are assigned accordingly.

Issues with test quality have been addressed by excluding test classes that have high failure rates from the CI testing build. Naturally, this does not solve the quality issues and therefore the failures in these classes have yet to be analyzed. Nevertheless, the build has currently very low number of false positives and it has also already detected some regression defects. Thus, the utilization of CI builds in UI test automation seems promising. Still, more work is required in order to take test quality in control.

All in all, the UI test automation process has taken a leap towards a more agile approach. Continuous and fast regression test results allow more rapid responses to detected defects. It is also acknowledged that the initial implementations in the scope of this thesis have issues that should be looked into. Thus, these implementations are one step on the road to developing the UI test automation tool and the process further.

5.4 Future thoughts and development ideas

This thesis concentrated on improving the UI test automation of M-Files Web. Moreover, an important theme was how these automated tests are utilized in continuous integration. In future, the suggestions that were not yet implemented in the scope of this thesis are worth considering.

One important aspect in the future is how the new CI build approach is adopted and developed further. For instance, it should be considered what role the original full testing round will take in this new approach. Additionally, the current build implementation is only testing M-Files trunk builds. Therefore, extending build coverage to different release branches, similar to the existing API test build, could be considered.

Some implementations presented in this thesis may also be applicable to other parts of M-Files test automation. For instance, M-Files Desktop UI test automation would also greatly benefit from an integration with a CI server. In fact, first steps have already been taken into that direction. On the other hand, both M-Files Desktop and API NUnit tests could also benefit from better utilization of concurrency. Effectively, utilizing multiple parallel builds in all types of test automation could also yield faster feedback. Naturally, fast feedback alone does not bring more value if no rapid actions are taken in response to it or if the feedback is ignored. It should also be noted that the duration for creating a new M-Files build is also something to look into. Faster building of M-Files could also enable more rapid feedback during manual user story testing.

On the other hand, test automation is now mainly utilized in the testing of the core M-Files product but exploring its possibilities in the customer specific implementation projects could also be considered. Automated regression testing performed on the solutions built on top of M-Files could help in assuring the quality of these customer deliveries.

Another idea inspired by continuous integration is the development of unit test by developers. These tests could exist on a lower implementation layer than the current M-Files API NUnit tests. Such tests often improve the quality of the base level code and emphasize the importance of good development practices. Naturally, the value of this practice should be assessed in the context of the current quality assurance process.

The quality assurance practices of M-Files are constantly developing and this thesis is one part in that continuous effort. It is important to keep developing testing practices in all areas and making sure that no important aspects are neglected.

6. CONCLUSIONS

The goal of this thesis was to improve UI test automation of M-Files Web by addressing identified issues in the testing process and in the testing tool. This thesis examined different practices in continuous integration, continuous testing, test automation, and user interface testing based on literature. This review revealed that quality of a software product is composed of several factors that must be assessed by different testing methods. User interface test automation integrated with a continuous integration build has its own role in rapid detection of regression defects and ensuring stable development builds.

The implemented Teamcity build and modifications to the user interface test automation tool have significantly reduced the testing duration and also automated most of the previously manual tasks. Additionally, the test results have become more visible, for instance, via the Teamcity dashboard and notifications. Moreover, the test quality has been taken into consideration in order to provide accurate test results in the continuous integration build. Finally, solutions to the monitoring of test duration have been examined and duration data is continuously collected but it is also acknowledged that the current implementation still does not provide well refined data on test case level. All in all, it can be stated that the thesis has met its goals because all the identified issues have either been solved or mitigated.

Overall, working with the user interface test automation and the continuous integration server has been an educational experience. However, when concentrating on automation solutions it is sometimes easy to lose focus on the main point: the quality of the product under test. Thus, it should always be assessed whether new or existing automation solutions really provide expected value towards product quality. Developing test automation on user interface level of different clients and on API level is a solid approach in regression testing of the product. Streamlining the test automation processes and improving the tools towards faster and more frequent feedback while emphasizing test quality seems like the correct direction.

The steps taken in the scope of this thesis mostly regarded tools and implementation choices in user interface test automation. The implemented improvements did not contain significant changes in the way of working in the organization. However, several of the suggestions that were not yet implemented also consider such changes. Thus, it should be remembered that improving the test automation practices should not only consider the used tools but also the responsibilities and working effort of the individuals. In the end, also the success of the implemented suggestions will become evident

only after the new build practices are put into real use in the product development process.

REFERENCES

Abdul, F.A., Fhang, M.C.S. (2012). Implementing Continuous Integration towards rapid application development. International Conference on Innovation, Management and Technology Research. pp. 118-123.

Angularjs.org. (2016). Angular.js website. https://angularjs.org/. Accessed on 19.5.2016.

Apache.org. (2016a). Apache Subversion. https://subversion.apache.org/. Accessed on 3.5.2016.

Apache.org. (2016b). Apache Maven Project. https://maven.apache.org/. Accessed on 16.1.2016.

Asp.net. (2016). ASP.NET website. http://www.asp.net/. Accessed on 19.5.2016.

Ariola, W. (2015). DevOps: Are You Publishing Bugs to Your Clients Faster? PNSQC 2015 Proceedings. Available: http://uploads.pnsqc.org/2015/papers/t-007_Ariola_paper.pdf. Accessed on 20.2.2016.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D. (2001). Manifesto for Agile Software Development. Available: http://agilemanifesto.org/. Accessed on 2.4.2016.

Brown, D. (2013). Agile User Experience Design, Morgan Kaufmann. 256 p.

Casteleyn, S., Daniel, F., Dolog, P., Matera, M. (2009). Engineering Web Applications, Springer-Verlag Berlin Heidelberg. 349 p.

Chacon, S., Straub, B. (2014). Pro Git, Apress. 574 p.

Cucumber.io. (2016). Cucumber website. https://cucumber.io/. Accessed on 25.5.2016.

Cunningham, W. (1992). The WyCash Portfolio Management System, OOPSLA '92 Experience report. Available: http://c2.com/doc/oopsla92.html. Accessed on: 12.5.2016.

Eclipse.org. (2016). Eclipse website. https://eclipse.org/. Accessed on 28.4.2016.

Enos, J. (2013). Automated builds: The Key to Consistency. InfoQ.com. Available: http://www.infoq.com/articles/Automated-Builds. Accessed on 23.11.2015.

Facebook.github.io. (2016). React website. https://facebook.github.io/react/. Accessed on 19.5.2016.

Fowler, M. (2003). TechnicalDebt. Available: http://martinfowler.com/bliki/TechnicalDebt.html. Accessed on 12.5.2016.

Fowler, M. (2006). Continuous Integration. Available: http://www.martinfowler.com/articles/continuousIntegration.html. Accessed on 22.11.2015.

Fowler, M. (2008). DslQandA. Available: http://martinfowler.com/bliki/DslQandA.html. Accessed on 12.12.2015.

Fowler, M. (2013). PageObject. Available: http://martinfowler.com/bliki/PageObject.html. Accessed on 19.5.206.

Galitz W.O. (2002). The Essential Guide to User Interface Design, Second Edition, Wiley. 784 p.

Getbootstrap.com. (2016). Bootstrap website. http://getbootstrap.com/. Accessed on 19.5.2016.

Git-scm.com. (2016). Git website. https://git-scm.com/. Accessed on 3.5.2016.

Gmeiner, J., Ramler, R., Haslinger, J. (2015). Automated Testing in the Continuous delivery pipeline: A case study of an online company. IEEE Eight International Conference on Software Testing, Verification and Validation Workshops. pp. 1-6.

Harty, J. (2011). Finding Usability bugs with Automated Tests. Acm.org. Available: http://queue.acm.org/detail.cfm?id=1925091. Accessed on 7.12.2015

Hass, A.M.J. (2008). Guide to Advanced Software Testing, Artech House. 459 p.

Humble, J., Farley, D. (2010). Continuous Delivery, Addison-Wesley. 497 p.

IEEE Computer Society. (2008). IEEE Std 829-2008. IEEE Standard for Software and System Test Documentation – Redline. pp. 161 p.

ISO/IEC/IEEE. (2010). ISO/IEC/IEEE 24765: 2010(E): Systems and software engineering -- Vocabulary. 418 p.

Jenkins-ci.org. (2015). Jenkins documentation. https://wiki.jenkins-ci.org/display/JENKINS/Use+Jenkins. Accessed on 20.12.2015.

Jetbrains.com. (2016a). Teamcity website. https://www.jetbrains.com/teamcity/. Accessed on 21.2.2016.

Jetbrains.com. (2016b). Teamcity documentation. https://confluence.jetbrains.com/display/TCD9/TeamCity+Documentation. Accessed on 5.2.2016.

Kaner, C. (2000). Architectures of Test Automation. Available: http://www.kaner.com/pdfs/testarch.pdf. Accessed on 20.2.2016.

Keith, J., Sambells, J. (2010). DOM Scripting Web Design with JavaScript and the Document Object Model, Appress. 336p.

Koch, A.S. (2004). Agile Software Development, Artech house Books. 302p.

Marick, B (1998). When Should a Test Be Automated? Available: http://www.stickyminds.com/sites/default/files/article/file/2014/When%20Should%20a%20Test%20Be%20Automated.pdf. Accessed on 20.2.2016.

McMahon, C. (2009). History of a Large Test Automation Project Using Selenium. AGILE '09 Agile Conference. pp. 363-368.

Melymuka, V. (2012). TeamCity 7 Continuous Integration Essentials, Packt Publishing Ltd. 128 p.

M-Files.com. (2016). M-Files website. https://www.m-files.com/en. Accessed on: 28.4.2016

M-Files Corporation. (2016a). Product Development PD-1094: Product Development v2.0.

M-Files Corporation. (2016b). R&D DoD for M-Files Core SW v1.

M-Files Corporation. (2016c). Product Development SOP-1143: Tracker Instructions v2.0

Myers, G.J., Sandler, C., Badgett, T. (2011). The Art of Software Testing (3rd Edition), Wiley. 254 p.

Nielsen, J. (1995). How to Conduct a Heuristic Evaluation. Available: https://www.nngroup.com/articles/how-to-conduct-a-heuristic-evaluation/. Accessed on 18.5.2016.

Nielsen, J. (2012). Usability 101: Introduction to Usability. Available: https://www.nngroup.com/articles/usability-101-introduction-to-usability/. Accessed on 17.5.2016.

Nodejs.org. (2016). Node.js website. https://nodejs.org/en/. Accessed on 19.5.2016.

Norman, D., Nielsen, J. (2016). The Definition of User Experience. Available: https://www.nngroup.com/articles/definition-user-experience/. Accessed on 17.5.2016.

NUnit.org. (2016). NUnit. http://www.nunit.org/. Accessed on 28.4.2016.

Rohrer, C. (2014). When to Use Which User-Experience Research Methods. Available: https://www.nngroup.com/articles/which-ux-research-methods/. Accessed on 18.5.2016.

Rubyonrails.org. (2016). Ruby on Rails website. http://rubyonrails.org/. Accessed on 19.5.2016.

Schwaber, K., Sutherland, J. (2013). The Scrum Guide. Available: http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-US.pdf. Accessed on 2.4.2016

SeleniumHQ. (2016). Selenium grid for selenium1 and webdriver. https://github.com/SeleniumHQ/selenium/wiki/Grid2. Accessed on 17.1.2016.

SeleniumHQ.org. (2016). Selenium documentation. http://www.seleniumhq.org/docs/index.jsp. Accessed on 16.1.2016.

Selenium.googlecode.com. (2016). Selenium API documentation. https://selenium.googlecode.com/git/docs/api/java/org/openqa/selenium/. Accessed on 16.1.2016.

SikuliX. (2016). SikuliX documentation basic info. Available: http://sikulix-2014.readthedocs.io/en/latest/basicinfo.html. Accessed on 25.5.2016.

SikuliX.com. (2016). SikuliX website. http://www.sikulix.com/. Accessed on 25.5.2016.

SmartBear.com. (2016). TestComplete overview. https://smartbear.com/product/testcomplete/overview/. Accessed on 28.4.2016.

Sonatype.org. (2016). The Central Repository. http://central.sonatype.org/. Accessed on 12.1.2016.

Stolberg, S. (2009). Enabling Agile Testing through Continuous Integration. AGILE '09 Agile Conference. pp. 369-374.

Techopedia.com. (2016a). Build definition. Available: https://www.techopedia.com/definition/3759/build. Accessed on 30.1.2016.

Techopedia.com. (2016b). A/B Test definition. Available: https://www.techopedia.com/definition/27398/ab-test. Accessed on 14.5.2016.

TestNG.org. (2015). TestNG documentation. http://testng.org/doc/documentation-main.html. Accessed on 20.12.2015.

Vmware.com. (2016). Vmware website: http://www.vmware.com/. Accessed on 1.5.2016.

Vuori, M. (2014). Support from testing for fast and dymamic software development. Project report. 21 p.

W3C. (2009). DOM. http://www.w3.org/DOM/. Accessed on 4.1.2016.

W3C. (2014) XMLHttpRequest API. http://www.w3.org/TR/XMLHttpRequest/. Accessed on 9.1.2016.

W3C. (2015). WebDriver documentation. http://www.w3.org/TR/2015/WD-webdriver-20151109/. Accessed on 7.12.2015.

WebPlatform.org. (2014). What is CSS?: http://docs.webplatform.org/wiki/tutorials/learning_what_css_is. Accessed on 4.1.2016.

WebPlatform.org. (2015). The Basics of HTML: http://docs.webplatform.org/wiki/guides/the_basics_of_html. Accessed on 4.1.2016.